

For explanations see se.ethz.ch/touch/

TOUCH OF CLASS

Learning to program well

**with Object Technology
and Design by Contract**

AN INTRODUCTION TO SOFTWARE ENGINEERING

Bertrand Meyer

Draft of ongoing work by Bertrand Meyer
TOUCH OF CLASS:

Learning to program well
with Object Technology and Design by Contract

Important: this text is in draft form and is intended for the use of students in the ETH Zurich “Introduction to Programming” course (252-001-00), as well as for members of the “Touch of Class” mailing list. Do not copy it other than for personal use, or distribute it without the author’s permission.

© Bertrand Meyer, 2003-2006

For a record of changes see [**“Change log”, page 595.**](#)

Short contents

The full table of contents appears on page [xxxvii](#).

Prefaces	v	PART IV: OBJECT-ORIENTED TECH-	
student_preface	vii	NIQUES	453
instructor_preface	xv	18 Inheritance	455
Contents	xxxvii	19 Operations as objects: agents and lambda calculus	457
PART I: BASICS	5	20 Event-driven design	501
1 The industry of pure ideas	5	21 Program correctness	535
2 Dealing with objects	17	PART V: TOWARDS SOFTWARE ENGI-	
3 Program structure basics	39	NEERING	537
4 The interface of a class	51	22 Overview of software engineering	539
5 Just Enough Logic	73	23 The software process	561
6 Creating objects and executing systems 109		24 Writing requirements and documentation	563
7 Control structures	137	25 Designing Graphical User Interfaces	565
8 Routines and functional abstraction	197	26 Testing and debugging	567
9 Variables, assignment and references	213	27 Towards software reuse	569
10 Fundamental data structures, genericity, and algorithm complexity	241	PART VI: APPENDICES	571
11 Input, output and exceptions	309	A Using the EiffelStudio environment	573
PART II: HOW THINGS WORK	311	B Eiffel syntax specification	577
12 Just enough hardware	313	C The C# language	579
13 Describing syntax	327	D The Java language	581
14 Programming languages	353	E The C language	583
15 Compilers and friends: the basic software tools	355	F The C++ language	585
PART III: ALGORITHMS AND DATA STRUCTURES	357	Picture credits	587
16 Recursion and trees	359	Index	589
17 An elegant algorithm family: Topological Sort	409	Change log	595

Prefaces

```
note
  description: "[
    This book has two prefaces, one for instructors and one for students, as stated
    here through a contrived but correct use of its own programming notation.
  "]"
class PREFACING inherit
  KIND_OF_READER
create
  choose
feature -- Initialization
  choose
    -- Get the preface that's right for you.
  do
    if is_student then
      student_preface.read
    elseif is_instructor then
      instructor_preface.read
    else
      pick_one_or_both
    end

    check
      -- You learn about dynamic binding
    note
      why: "You'll express this more elegantly"
    end
  end
end
```

*student_preface**

*The preface for instructors is on page [xv](#).

Programming is fun. Where else can you spend your days devising machines of your own imagination, build them without ever touching a hammer or staining your clothes, make them run as by magic, and get paid — not too bad, thanks for asking — at the end of the month?

Programming is tough. Where else do products from the most prestigious companies fail even in ordinary use? Where else does one find so many users complaining so loudly? Where else do engineers routinely work for hours or days trying to understand why something that should work doesn't?

Get ready for the mastery of programming and its professional form, software engineering; get ready for both the toughness and the fun.

Software everywhere

By going into computing science you have chosen one of the most exciting and fast-moving topics in science and technology. Fifty years ago it wasn't even recognized as a scientific subject; today there's hardly a university in the world without a CS department. Thousands of books, conferences, journals, magazines cover the field. The worldwide revenues of its industry — called information technology or IT — are in the trillions. It is hard to think, in the history of technology, of any field that has undergone growth of either such magnitude or such speed.

And we've made a difference. Without software there would be no large-scale plane travel, and in fact no modern planes (or modern cars, or high-speed trains) since their design requires sophisticated "Computer-Aided Design" software. To pay its employees, any large corporation would employ hundreds of people just to write the paychecks. There would be no video games, camcorders, iPods; a phone would still be a device requiring a cable to a wall outlet; to produce a report we would still hand-write a draft, give it over to a professional typist, and go through rounds of copy-editing. A sudden itch

to know the name of the captain in *The Grand Illusion*, or the population of Cape Town, or the author of a certain familiar citation, would require (rather than typing three words and getting the answer in a second) a trip to the library. The list goes on; at the heart of countless practices that now affect our daily life lie programs — increasingly sophisticated programs.

Through this book you will become familiar with the world of programs and programming, with a view to becoming a professional in the field.

Casual and professional software development

Although more and more people are acquiring basic computing proficiency, ability to do computing at a professional level is another matter, and it's what a curriculum in computing science will bring you.

For comparison, consider mathematics. A few centuries ago, just being able to add and subtract 5-digit numbers required a university education, and in return provided qualifications for such good jobs as accountant. Nowadays these skills are taught in grade school; if you want to become an engineer or a physicist, or just a stock trader, you need to study more advanced mathematical topics, such as calculus, in a university. The boundary between basic training and university-level education has moved up.

Computing is following the same evolution, only much faster — the scale is decades, not centuries. Not so long ago, being somehow able to program a computer was enough to land a job. Don't expect this today; an employer will not be much more impressed if your résumé states "I have written programs" than if you say you can add numbers.

What increasingly counts is the difference between having some basic programming experience and being a software engineer. The former skill will soon be available to anyone who has gone through a basic education; but the latter is a professional qualification, just like advanced mathematics. Studying this book is a step towards becoming a computing professional.

If you have done some computing before, you will recognize some of the ideas, but you should also expect to be surprised at times, since the professional study of any topic is different from its use by the general public. Once in a while, for example, you may find that I belabor a seemingly simple matter. If so, you will (I think) discover after a while that the topic is *not* as simple as it seems at first — just as addition is more challenging to the mathematician than to the accountant.

Factors that distinguish professional software development from casual programming include **size**, **duration** and **change**. In professional software development, you may become involved in programs that reach into the millions of lines of program text, must remain in operation for years or decades, and will undergo many changes and extensions in response to new

circumstances. Many a problem that seems trivial or irrelevant when you are working on a medium-size program, only meant to solve a problem of immediate interest, becomes critical when you move to the scale of professional development.

With this book I'll try to prepare you for the real world of software, where systems are complex, solve serious problems (often affecting human life or property), stay around for a long time, and must lend themselves gracefully to requests for change.

Prior experience — or not

This book doesn't assume any prior experience in programming.

If you *did* program before, that experience will help you master some of the concepts faster. While you must be prepared to question some of your previous practices if they do not match the professional software engineering principles developed here, you can and should take advantage of everything you know. Learning to program well takes a lot of effort: every bit — every angle from which you can approach the problem — helps. In particular, the discussion relies, as explained below in more detail, on a supporting software system, Traffic. If you are familiar with programming and some programming languages, you will be able to discover some of Traffic by yourself, possibly ahead of the official assignments. Don't hesitate to do so: one learns programming in part by reading existing programs for inspiration and imitation. You may have to do some guessing for elements of Traffic that rely on techniques and language constructs you haven't formally studied yet, but this is where your experience will help you move faster.

On the other hand, if you have *not* done any programming, you're OK too. You might progress more slowly at the beginning, but should just study all the material carefully and do all the exercises. In particular, even though there's little actual mathematics in this book, it particularly helps if you have a mathematical mindset and the practice of logical reasoning. This is just as beneficial as programming experience, and will compensate any handicap you feel relative to those fellow students who look like they typed in their first program before they lost their baby teeth.

Programming, like the rest of computing science, is at the confluence of engineering and science. Success requires both a hands-on attitude (the "hacker" side, in the positive sense of the word), useful in technology-oriented work, and an aptitude to perform abstract, logical reasoning, required in mathematics and other sciences. Experience with programming helps you with the first goal; a logical mind helps you with the second. Wherever your strength lies, take advantage of it, and use this book to compensate for any initial deficiency on the other side.

Modern software technology

Becoming a software professional requires more than one course or one book: it takes a multi-year curriculum in which you will learn about software engineering, theory of computation, data structures, algorithms, operating systems, artificial intelligence, databases, hardware, networking, project management, software metrics, numerical computation, graphics and many other topics. But to prepare you for these other courses it is essential to use the best of what is known in software technology.

In recent years two major ideas, holding the potential for producing software of much better quality than was available before, have made their way into the software field: **object-oriented software construction** and **formal methods**. Both of these ideas, but especially the first, can be used to make the introductory study of computing more exciting and more profitable. Along with other concepts from modern software technology, they play a major role in this book. Let's have a quick look at both of them.

Object-oriented software construction

Object-oriented (“O-O”) software construction follows from the realization that proper systems engineering requires the ability to rely on a large inventory of high-quality reusable components, as in the electronic or construction industries. The O-O approach defines what form these components should have: each of them must be based on a certain *type of objects*. The term “object”, which gives its name to the method, does not just refer to objects of the application domain, for example circles or polygons in a graphics program, but also to objects that are purely internal to the software, such as a list. If you do not quite see what this all means, that's normal; I hope that if you read this Preface again in a few months it will all be crystal-clear!

Object technology (the shorter name for object-oriented software construction) is quickly changing the software industry, and becoming familiar with it from the very beginning of your computing studies is an excellent insurance policy against technical obsolescence.

Formal methods

Formal methods are the application of systematic reasoning techniques, based on mathematical logic, to the construction of reliable software. Reliability, or rather the lack of it, is a vexing problem in software; errors, or the fear of error, are the programmer's constant companion. Anyone who uses computers has some anecdote about bugs.

Formal methods can help improve this situation. To learn formal methods in their full extent requires more knowledge than is available at the beginning of a university education. But the approach used in this book shows a significant influence of formal methods, in particular through the idea of *Design by Contract* which considers the construction of software systems as the implementation of a number of individual contractual relations between modules, each characterized by a precise specification of obligations and benefits. I hope that you will understand the importance of these ideas and remember them for the rest of your career; in industry, everyone knows the difference between a programmer who just “hacks code” and one who is able to produce correct, robust, durable software elements.

Learning by doing

This book is not a theoretical presentation; it assumes that as you go along you practice what you learn on a computing system. The associated [Web site](http://www.touch.ethz.ch) *touch.ethz.ch* provides the necessary software, in versions for Windows, Linux and MacOS, which you can download. Your school may also have the equivalent facilities available on its computers. In fact, the material is so organized as to prompt you, in some cases, to do the practical work with the software *before* learning the theoretical concepts.

The system that you will use for this course is one of the major object-oriented environments: EiffelStudio, an implementation of the Eiffel analysis, design and programming language. Eiffel is a simple, modern language, used worldwide in large, mission-critical industrial projects (banking and finance, health care, networking, aerospace etc.) as well as for teaching and research in universities. The EiffelStudio version that you will use is exactly the same as the professional version, with the same graphical development environment and fundamental reusable components such as the EiffelBase, EiffelVision and EiffelMedia libraries. Your school may also have an academic license providing for maintenance and support.

An appendix presents an introduction to four others languages also widely used in industry: C#, Java, C and C++.

From the consumer to the producer

Because from day one of the course you will have the whole power of EiffelStudio at your fingertips, you'll be able to skip many of the “baby” exercises that have traditionally been used to learn programming. The approach of this book is based on the observation that to learn a technique or a trade it is best to start by looking at the example of excellent work produced by professionals, and taking advantage of it by (in order) using that work, understanding its internal construction, extending it, improving it — and starting to build your own. This is the time-honored method of apprenticeship, which places newcomers under the guidance of experts.

The expertise is represented here by software, more specifically *library classes*: software elements from the Traffic library, specially developed for this book. As you write your first software examples, you will use these classes to produce results which are already impressive even though you haven't had much to write; you will be just relying on the mechanisms defined by the classes, acting, through your own software, as a *consumer* of existing components. Then, as someone who knows how to drive but is studying to become an automobile engineer, you will be encouraged to lift the hood and see how these classes are made, so that you can later on write extensions to the classes, improve them perhaps, and write your own classes.

The Traffic library, as its name suggests, provides mechanisms for dealing with traffic in a city — cars, pedestrians, metros, trams... —, with graphical visualization, simulations, computing routes, animating the routes etc. It's a rich reservoir of applications and extensions: you can use it to build video games, solve optimization problems, learn many new algorithms.

The built-in examples use Paris as the sample city, because it's a well-known tourist destination; you can easily adapt them to another city without touching the Traffic software, since all the location information is provided separately in a file (using a standard format, XML). It suffices to provide such a file representing your chosen city. For example, the course is taught at ETH Zurich with examples representing the Zurich tram system, replacing the Paris metro network.

Abstraction

Basing your work on existing components has another important consequence for your education as a professional software engineer. The program modules that you reuse are a substantial piece of software, embodying a lot of knowledge. It would be very difficult to use them for your own applications if you had to read the full program text of each one you need. Instead, you will rely on a description of their *abstract interfaces*, which are extracted from

their text (by automatic software mechanisms, part of EiffelStudio) but retain only the essential information that, as a consumer, you need. An abstract interface is a description of the purpose of a software module, which states only its functions, not *how* the module's code realizes these functions. In software terms, it's also called the *specification* of the module, excluding the module's *implementation*.

This technique will help you learn a key skill of the professional software developer: *abstraction*, meaning here the ability to distinguish the purpose of any piece of software from the details, often numerous, of its implementation. Every professor and textbook of software development preaches the virtues of abstraction, and for good reason; here you'll get the occasional bit of preaching too but mostly you'll been encouraged to learn abstraction by example, experiencing its benefits through the reuse of existing components. When you get to build your own software you should apply the same principles; that's the only way to tame the ogre of software complexity.

The benefits of abstraction are quite concrete; you'll be able to experience them right from the beginning. The first program you'll write is only a few lines long, but already produces a significant result (an animated itinerary on a city map). It can do this only by using modules from Traffic; and it can use them only because they are available through an abstract specification. If you had to examine the text of these modules (their *source code*), then the text of the modules they rely on themselves, and so on, you would quickly drown in an ocean of details and could not produce anything.

→ "[A CLASS TEXT](#)",
[2.1, page 17](#).

Throughout your work with software, abstraction is the lifevest that will save you from drowning in the sea of complexity.

Destination: quality

This book teaches not only techniques but methodology. Throughout the presentation you will encounter design principles and rules on programming style. Sometimes you may think I'm being fussy and that you could write the program just as well without the rules. Well, often you can. But the methodological rules make the difference between an amateurish program, which sometimes works, sometimes not, and the kind of professional-quality software that you will want to produce. You should apply these rules not just because this book and your teachers say so, but because the power and speed of computers magnify any deficiency, however small, and requires from the programmer attention both to the big picture and to every detail. They are also good job insurance for your future career: there are many programmers around, and what really differentiates them in the eyes of an employer is the long-term quality of the software they produce.

Don't fool yourself with the excuse that "this is only an exercise" or "this is only a small program":

- Exercises are precisely where you need to learn the best possible techniques; when Airbus hires you to write the control software for their next plane, it will be too late.
- Calling a program "small" is often more hope than guarantee. In industry, many big programs are small programs that grew, since a good program tends to give its users endless ideas for requesting new functionalities.

So you should apply the same methodological principles throughout the programs you develop, whether small or large, educational or operational.

This is the goal of this book: not just to take you through the basics of software engineering and to let you experience the fun and thrill of producing software that works, but also to develop — along with a sense of beauty for the principles, methods, algorithms, data structures and other techniques that define the discipline — a sense for what makes good software stand out, and a determination to produce programs of the highest possible quality.

BM

Zurich / Santa Barbara, October 2006

*instructor_preface**

*The preface for students is on page [vii](#).

Right from its subtitle, this book shows its colors: it's not just about learning to program but about “Learning to Program *Well*”. I am trying to get the students started on the right track so that they can enjoy programming — without enjoyment one doesn't go very far — and have a successful career; not just a first job, but a lifelong ability to tackle new challenges.

To help them reach this goal, the book applies innovative ideas detailed below:

- **Inverted curriculum**, also known as the “outside-in” approach, relying on a large library of reusable components.
- Pervasive use of **object-oriented** and model-driven techniques.
- **Eiffel** and **Design by Contract**.
- A moderate dose of **formal methods**.
- Inclusion, from the very beginning, of **software engineering** concerns.

These techniques have for several years been applied to the “Introduction to Programming” course at ETH Zurich, taken by all entering Computer Science students. *Touch of Class* builds on this course and draws from its lessons. This also means that teachers using it as a textbook can rely on the [teaching material](#) developed for the course: a full set of slides, lecture schedules, exercises, self-study tutorials, student projects, even video recordings of our lectures.

See se.ethz.ch/touch.
Most material in English; some German versions available.

THE CHALLENGES OF A FIRST COURSE

Many computer science departments around the world are wondering today how best to teach introductory programming. This has always been a difficult task, but new challenges have added themselves to the traditional ones:

This section is based on [\[12\]](#).

- Adapting to ever higher stakes.
- Identifying the key knowledge and skills to teach.
- Coping with fads and outside pressures.
- Addressing a broad diversity of initial student backgrounds and abilities.
- Meeting high expectations for examples and exercises.
- Introducing the real challenges of professional software development.
- Teaching methodology and formal techniques without scaring off students.

The stakes are getting ever higher. In educating future software professionals, we must teach durable skills. It is not enough to present immediately applicable technology, for which in our globalized industry a cheaper programmer will always be available elsewhere.

We must **identify the key knowledge and skills** to teach. Programming is no longer a rare, specialized ability; a large proportion of the population gets exposed to computers, software and some rudimentary form of programming, for example through spreadsheet macros or Web site development with Javascript, PHP or ASP.NET. Software engineers need more than being able to program; they must master software development as a professional endeavor, and by this distinguish themselves from the occasional or amateur programmer.

It is important to keep a cool head in the presence of **fads and outside pressures**. Fads are a given of our field, and they are not always bad — structured programming, object technology and design patterns were all fads once — but we must make sure an idea has proved its worth before inflicting it on our students. Outside pressures can be more delicate to handle. Student families have more say nowadays than in the past; this too is not necessarily a bad thing, but sometimes results in inappropriate demands that we teach the specific technologies required in the job ads of the moment. What this attitude misses is that four years later some of the fashionable acronyms will be different, and that good industry recruiters look for problem-solving skills, not narrow knowledge. It is our duty to serve the very interests of the students and their families by teaching them the fundamental matters, which will give them not just a first job but a rewarding career.

This whole obsession with learning the right résumé-filling buzzwords for fear of not landing a job is silly anyway. It is a worldwide phenomenon, likely to last for decades, that a decent software developer has no trouble finding a good job. For all the gloom that the media have spread after the “burst of the Internet bubble”, and the fears that “all the jobs have gone to Bangalore”, no end is in sight to the challenges and excitement of our field, including of course for our colleagues in Bangalore. But there’s a qualification: people who get and *keep* good jobs are not the narrow-minded specialists having been taught whatever filled the headlines of the day; they are the professional developers with a wide and deep understanding of computing science, and mastery of many complementary technologies.

The **broad diversity of student backgrounds** complicates the task. Among the students in the lecture hall on the first day of the typical introductory course, you may find some who have barely touched a computer, some who have already produced an e-commerce site, and the full range in-between. What can the teacher do?

- It is tempting to assume a fair amount of prior programming experience and teach to the most advanced students only; but this shuts out students who simply haven't had the opportunity or inclination to work with computers yet. In my experience, they include some who later turn out to be excellent computer scientists thanks to excellent abstraction skills, which they have so far applied to topics such as mathematics rather than computing. The nerdy image still widely associated with computers may have prevented them from realizing that it's not about late-night video game sessions fueled by home-delivery pizza (a picture which, in particular, turns off many girls with excellent computer science potential) but about cogent thinking applied to solving some of the most exciting intellectual challenges open to humankind.
- We must not either — at the other extreme — bring everyone down to the lowest level: we need a way to catch and retain the attention of the more experienced students, letting them use and expand the insights they have already gained.

Reliance on reusable components, discussed below, is a central part of this book's solution to the issue. By giving students access to high-quality libraries, we let the novices take advantage of their functionality through abstract interfaces without needing at first to understand what's inside. The more advanced and curious students can, ahead of the others, start to peek into the internals of the components and use them as guidance for their own goals.

For this to work we need **high-quality examples**. Students today, having lived most of their lives in a world awash in the visual and auditive marvels of software-powered multimedia, expect to see and build more than small academic programs of the “Compute the 7-th Fibonacci number” kind. We must meet these expectations of the “Nintendo Generation” [3], without of course letting technological dazzle push aside the teaching of timeless skills.

A variant of this issue is what we may call the “Google-and-paste” phenomenon, the name I use for what colleagues (generally using Java or C++ as the teaching language) report as follows: you give an exercise that calls for, say, a 100-line program solution. Internet-savvy students quickly find on the Web some Java code that does the job, except that it does much more as part of, maybe, a 10,000-line program. Now it doesn't take long for beginners to hit upon a key piece of programming wisdom from the ages: that if you see a program that works you mess with it as little as you can. You hold your breath when coming anywhere close to it. Following this insight, the student will just switch off (rather than remove) the parts he or she doesn't need, through a minimal set of changes. So the teacher gets a 10,000-line solution to an elementary question. Of course one may

impose, if not a full prohibition of Web use (which in a computer science curriculum would sound bizarre), precise rules that would exclude such a “solution”. But how exactly? “Google-and-paste” is, after all, a form of reuse, even if not exactly the kind advocated by software engineering textbooks.

The approach used in this book goes one step further. Not only do we encourage reuse, we actually provide a large amount of code (150,000 lines of Eiffel at the time of writing, and growing) for reuse, and also for imitation since it is available in source form and explicitly designed as a model of good design and implementation. Reuse is from the beginning one of the “best practices” promoted by the course; but it’s a form of reuse in line with principles of software engineering, based on abstract interfaces and contracts.

These questions contribute to the next issue on our list: **introducing the real challenges of professional software development**. In a university-level computer science or software engineering program, we can’t just teach programming in the small. We have to prepare students for what matters to professionals: programming in the large. Not all techniques that work well for small programs will scale up. The very nature of an academic environment, especially at an introductory level, makes it hard to introduce students to the actual challenges of today’s industrial software: software developed by many people, expanding to many lines of code, adapted to many categories of uses and users, maintained over many years, undergoing many changes.

This concern for scalability gives particular urgency to the last issue: **introducing methodology and formal reasoning without disconnecting from the students**. Methodological advice — to use abstraction, information hiding, contracts and software engineering principles in general — can sound preachy and unnecessary. Introducing some formal (mathematically-based) techniques, such as the notion of loop invariant, can widen this potential gap between teacher and student. Paradoxically, the students who have already programmed a bit and stand to benefit most from such admonitions and techniques may be most tempted to discard them since they know from experience that it is possible — on small programs! — to reach an acceptable result without strict rules.

The best way to instill a methodological principle is pragmatic: to show that it empowers you to do something that would otherwise be unthinkable, such as building impressive programs with graphics and animation. Our reliance on powerful libraries of reusable components is an example: students can, right from the beginning of the course, produce significant applications, visual and all, thanks to these components; but they would never proceed beyond a few classes if as a prerequisite they had to read the code. The only reuse that works here is through abstract interfaces.

Rather than pontificating on abstraction, information hiding and contracts, it is better to let the students use these techniques and discover that they work. If an idea has saved you from drowning, you won’t discard it as futile theoretical advice.

OUTSIDE-IN: THE INVERTED CURRICULUM

The order of topics in programming courses has traditionally been bottom-up: start with the building blocks of programs such as variables and assignment; continue with control and data structures; move on if time permits — which it often doesn't in an intro course — to principles of modular design and techniques for structuring large programs.

This approach gives the students a good practical understanding of the fabric of programs. But it may not always teach the system construction concepts that software engineers must master to be successful in professional development. Being able to produce programs is not sufficient any more today; many people who are not professional software developers can do this honorably. What distinguishes the genuine professional is the mastery of system skills for the development and maintenance of possibly large and complex programs, open for adaptation to new needs and for reuse of some of their components. Starting from the nuts and bolts, as in the traditional “CS1” curriculum, may not be the best way to teach these skills.

Rather than bottom-up — or top-down — the order of this book is **outside-in**. It relies on the assumption that the most effective way to learn programming is to use good existing software, where “good” covers both the quality of the code — since so much learning happens through imitation of proven models — and, almost more importantly, the quality of its *interfaces*, in the sense of program interfaces (APIs).

From the outset we provide the student with powerful software: a set of sophisticated libraries, called Traffic, where the top layers have been produced specially for this book, and the basic layers on which they rely (data structures, graphics, GUI, time and date, multimedia, animation...) are widely used in commercial applications. All this library code is available in source form, providing a repository of high-quality models to imitate; but in practice the only way to use them for one's own programs, especially at the beginning, is through API specifications, also known as *contract views*, which provide the essential information abstracted from the actual code. By relying on contract views, students are able right from the start to produce interesting applications, even if the part they write originally consists of just a few calls to library routines. As they progress, they learn to build more elaborate programs, and to understand the libraries from the inside: to “open up the black boxes”. The hope is that at the end of the course they will be able, if needed, to produce such libraries by themselves.

This Outside-In strategy results in an “Inverted Curriculum” where the student starts as a *consumer* of reusable components and learns to become a *producer*. It does not ignore the teaching of standard low-level concepts and skills, since at the end we want students who can take care of everything a

program requires, from the big picture to the lowest details. What differs is the order of concepts and particularly the emphasis on architectural skills, often neglected in the bottom-up curriculum.

The approach is intended to educate students so that they will master the key concepts of software engineering, in particular *abstraction*. In my career in industry I have repeatedly observed that the main quality that distinguishes good software developers is their ability to abstract: to separate the essential from the accessory, the durable from the temporary, the specification from the implementation. All good introductory textbooks duly advocate abstraction, but the result of such exhortations is limited if all the student knows of programming is the usual collection of small algorithmic examples. I can lecture on abstraction too, but in the end the most effective way to convey the concepts is by example; by showing to the student how he or she can produce impressive applications through the reuse of existing software. That software is large at least by academic standards; trying to reuse it by reading the source code would take months of study. Yet students can, in the first week of the course, produce impressive results by reusing it through the contract views.

Here abstraction is not just a nice idea that we ask our students to heed, another parental incitation to be good and do right. It's the only way to survive when faced with an ambitious goal that you can't reach except by standing on someone else's shoulders. Students who have gone early and often through this experience of building a powerful application through contract-based reuse of libraries do not need much more haranguing about abstraction and reuse; for them these concepts become a second nature.

Teaching is better than preaching, and if something is better than teaching it must be the demonstration — carried out by the students themselves — of the principles at work, and the resulting “Wow!”.

THE SUPPORTING SOFTWARE

Central to the Outside-In approach of this book is the accompanying Traffic software, [available](#) for free download. The choice of application area for the library required some care:

From touch.ethz.ch.

- The problem domain should be immediately familiar to any student, so that we can spend our time studying software issues and solutions, not the problem domain. (It might be fun to take, say, astronomy, but then we'd end up discussing comets and galaxies rather than inheritance structures and class invariants.)

- The area should provide a large stock of interesting algorithm and data structure examples, applications of fundamental computer science concepts, and new exercises that each instructor can devise beyond those in the book. This should extend beyond the introductory course, to enable our colleagues teaching algorithms, distributed systems, artificial intelligence and other computer science topics to take advantage of the software if they wish.
- The chosen theme should call for graphics and multimedia development as well as advanced graphical user interfaces.
- Unlike many video games, it must not involve violence and aggression, which would be inappropriate in a university setting (and also would not help correct the gender imbalance which plagues our field).

The application area that we retained is *transportation in a city*: modeling, planning, simulation, display, statistics. The supporting Traffic software is not just an application, doing a particular job, but a *library*, providing reusable components from which students and instructors can build applications. Although still modest, it has the basic elements of a Geographical Information System and the supporting graphical display mechanisms.

For its examples the book uses Paris, with its streets and transportation networks; since the city's description comes from XML files, it is possible to retarget the example to any other city. (In the first session of the course at ETH a few students spontaneously provided a file representing the Zurich transportation network, which we have been using ever since.)

The very first application that the student produces takes up four lines of code. Its execution displays a map, highlights the Paris Metro network on the map, retrieves a predefined route, and shows a visitor traveling that route through video-game-style graphical animation. The code is:

```
class PREVIEW inherit
  TOURISM
feature
  explore is
    -- Show city info and route.
  do
    Paris.display
    Louvre.spotlight
    Metro.highlight
    Route1.animate
  end
end
```

The algorithm takes up four lines of code, and yet its effect is impressive thanks to the underlying Traffic mechanisms.

In spite of the reliance on an extensive body of existing software, I stay away from giving any impression of “magic”. It’s indeed possible to explain everything, at an appropriate level of abstraction. I never say “*just do as you’re told, you’ll understand when you grow up*”. This attitude is no better at educating students than it is at raising your children. In the first example as shown above, even the **inherit** clause can be explained in a simple fashion: I don’t go into the theory of inheritance, of course, but simply tell the student that class *TOURISM* is a helper class introducing predefined objects such as *Paris*, *Louvre*, *Metro* and *Route1*, and that a new class can “inherit” from such an existing class to gain access to its features. They’re also told that they don’t need to look up the details of class *TOURISM*, but may do so if they feel the engineer’s urge to know “how things work”.

The rule, allowing our students to approach the topics progressively, is always to abstract and never to lie.

From programming to software engineering

Programming is at the heart of software engineering, but is not all of it. Software engineering concerns itself with the production of systems that may be large, are developed over a long time, undergo many changes, and meet strong constraints of quality, timeliness and cost. Although the corresponding techniques are usually not taught to beginners, it’s really important to provide at least a first introduction. The topics include debugging and testing (even with the best of modern programming methodology, this will account for a good deal of the time spent on the job), quality in general, lifecycle models, requirements analysis (the programmers we are educating shouldn’t just be techies focused on the machinery but should also be able to talk to users and understand their customers’ needs), GUI design.

Terminology

Lucid thinking includes lucid use of words. I have devoted particular attention to consistent and precisely defined terminology. The most important definitions appear in call-out boxes, others in the main body of the text.

At the end of each chapter a “New vocabulary” section lists all the terms introduced, and the first exercise asks the student to provide precise definitions of each. This is an opportunity to test one’s understanding of the ideas introduced in the chapter.

TECHNOLOGY CHOICES

The book relies on a combination of technologies: an object-oriented approach, Design by Contract, Eiffel as the design and programming language. It is important to justify these choices and explain why some others, such as Java as a programming language, were not retained.

Object technology

Many introductory courses now use an object-oriented language, but not necessarily in an object-oriented way; few people have managed to blend genuine O-O thinking into the elementary part of the curriculum. Too often, for example, the first programs rely on static functions (in the C++ and Java sense of routines not needing a target object). There sometimes seems to be an implicit view that before being admitted to the inner chambers of modern technology students must suffer through the same set of steps that their teachers had to travel in their time. This approach continues the traditional bottom-up order of concept introduction, reaching classes and objects only as a reward to the students for having patiently climbed the *Gradus ad Parnassum* of classical programming constructs.

There's no good reason for being so coy about O-O. After all, part of the pitch for the method is that it lets us build software systems as clear and natural *models* of the concepts and objects with which they deal. If it's so good, it should be good for everyone, beginners included. Or to borrow a slogan from the waiters' T-shirts at Anna's Bakery in Santa Barbara, whose coffee played its part in fueling the writing of this book: *Life is uncertain — Eat dessert first!*

Classes and objects appear indeed at the very outset and serve as the basis for the entire book. I have found that beginners adopt object technology enthusiastically provided the concepts are introduced, without any reservations or excuses, as the normal, modern way to program.

One of the principal consequences of the central role of object technology in this presentation is that the notion of *model* guides the student throughout. The emergence of “model-driven architecture” reflects the growing recognition of an idea central to object technology: that successful software development relies on the construction of models of physical and conceptual systems. Classes, objects, inheritance and the associated techniques provide an excellent basis to teach effective modeling techniques.

Object technology is not exclusive of the traditional approach. Rather, it subsumes it, much as relativity yields classical mechanics as a special case: an O-O program is made of classes, and its execution operates on objects, but the classes contain routines, and the objects contain fields on which programs may operate as they would with traditional variables. So both the *static* architecture of programs and the *dynamic* structure of computations cover the traditional

concepts. We absolutely want the students to master the traditional techniques such as algorithmic reasoning, variables and assignment, control structures, procedures and recursion, and to be able to build entire programs from scratch.

Eiffel and Design by Contract

We rely on Eiffel and the EiffelStudio environment which students can download for free from www.eiffel.com. Universities can also install this free version (and purchase support if desired). This choice directly supports the pedagogical concepts of this book:

- The Eiffel language is uncompromisingly object-oriented.
- Eiffel provides a strong basis to learn other programming languages such as C#, Java, Smalltalk or C++.
- Eiffel is easy to learn for a beginner. The concepts can be introduced progressively, without interference between basic constructs and those not yet studied.
- The EiffelStudio development environment uses a modern, intuitive GUI, with advanced facilities including sophisticated browsing, editing, debugging, automatic documentation (HTML or otherwise), even software metrics. It produces architectural diagrams automatically from the code and, the other way around, lets a user draw diagrams from which the environment will produce the code, with round-trip capabilities.
- EiffelStudio is available on many platforms including Windows, Linux, Solaris, Microsoft .NET, Mac OS X.
- EiffelStudio includes a set of carefully written libraries, which support the reuse concepts of this book, and serve as the basis of the Traffic library written specifically for it. The most relevant are *EiffelBase*, which by implementing the fundamental structures of computer science supports the study of algorithms and data structures in part [III](#), *EiffelTime* for date and time, *EiffelVision*, an advanced portable graphical library, and *EiffelMedia* for multimedia and animation facilities.
- Unlike tools designed for education only, Eiffel is used commercially for large mission-critical applications handling billions of dollars of investment, managing health care, performing civil and military simulations, and tackling other problems across a broad range of application areas. This is in my opinion essential to effective teaching of programming; a tool that is really good should be good for professionals as well as for novices.
- The Eiffel language is specified by a standard of the International Standards Organization. For the teacher relying on a programming language, an international standard, especially an ISO standard, is a guarantee of sustainability and precise definition.

For the text of the standard see tinyurl.com/y5abdx or the ECMA version (same contents) at tinyurl.com/cq8gw.

- Eiffel is not just a programming language but a *method* whose primary aim — beyond expressing algorithms for the computer — is to support *thinking* about problems and their solutions. It enables us to teach a **seamless approach** that extends across the software lifecycle, from analysis and design to implementation and maintenance. This concept of seamless development, supported by the round-trip Diagram Tool of EiffelStudio, is in line with the modeling benefits of object technology.

To support these goals, Eiffel directly implements the concepts of **Design by Contract**, which were developed together with Eiffel and are closely tied to both the method and the language. By equipping classes with preconditions, postconditions and class invariants, we let students use a much more systematic approach than is currently the norm, and prepare them to become successful professional developers able to deliver bug-free systems.

One should also not underestimate the role of syntax, for beginners as well as for experienced programmers. Eiffel's syntax — illustrated by the short example above — facilitates learning, enhances program readability, and fights mistakes:

- The language avoids cryptic symbols.
- Every reserved word is a simple English word, unabbreviated (*INTEGER*, not *int*).
- The equal sign `=`, rather than doing violence to hundreds of years of mathematical tradition, means the same thing as in math.
- In most of today's languages, program texts are peppered with semicolons terminating declarations and instructions. Most of the time there is no reason for these pockmarks; even when not consciously noticed, they affect readability. Being required in some places and illegal in others, for reasons not always clear to beginners, they can be a source of errors. In Eiffel the semicolon as separator is optional, regardless of program layout. This leads to a neat program appearance, as you may see by picking any example in the book.

Encouraging such cleanliness in program texts should be part of the teacher's pedagogical goals. Eiffel includes precise style rules, explained along the way to show students that good programming requires attention to both the high-level concepts of architecture and the low-level details of syntax and style: quality in the large and quality in the small.

More generally, a good language should let its users focus on the concepts rather than the notation. This is one of the goals of using Eiffel for teaching: that students should think about their problems, not about Eiffel

Why not Java?

Since courses in recent years have often used Java, or a Java variant such as C#, it is appropriate to explain why we don't follow this practice. Java is useful for computer scientists to know — indeed the book provides an appendix that introduces the basics of Java, and others on C#, C++ and C — but not appropriate as a first teaching language. There is simply too much baggage to be learned before the student can start to think about the problems. This is apparent from the first program attempts; a Java “Hello World” reads

```
class First {  
    public static void main(String args[])  
    { System.out.println("Hello World!"); } }
```

This is full of irrelevant concepts, each an obstacle to learning. Why “**public**”, “**static**”, “**void**”? (Sure, I'll make my program *public* if you insist, but do you mean my efforts are *void* of any value?) These keywords have nothing to do with the purpose of the program, and the student won't begin to understand what they mean for a few months at least, yet he or she must include them, like magic incantations, for their programs to work. For the teacher this means engaging far too often in injunctions to use certain constructions without understanding what they mean. As noted above this “*You'll understand when you grow up*” style is not good pedagogy. Eiffel protects us from it: we can explain every programming language construct that we use, right from the first program example.

The object-oriented nature of Eiffel and the simplicity of the language play a role. It is ironic that every Java program, starting with the simplest example as shown above, uses as main program a **static** function, that is to say a departure from the object-oriented style of programming. There are of course people who don't like the idea of using an O-O approach for the first programming course; but if you do go for objects, you should use them all the way. Trying to explain the O-O style to students cannot be very effective if you have to reveal to them — when they progress enough to understand the first thing they had to write, and repeated in every subsequent example — that this key part of all Java programs is not object-oriented after all.

Syntax, as noted, matters. In this first example the student must master strange symbol accumulations, like the final “**); } }**” above, disconcerting to the eye and with no obvious role. In this accumulation the precise order of the symbols is essential, but is hard to explain and to remember. (Why a semicolon between a closing parenthesis and a brace? Is there a space after that semicolon and if so how important is it?) Such aspects are troubling to

beginners; inevitably, much time and effort are consumed learning them and recovering from trivial mistakes with mysterious results, just when the student should be concentrating on the concepts of programming.

Another source of confusion is the use of “=” for assignment, inherited from Fortran through C and hard to justify in the twenty-first century. How many students starting with Java have wondered what value a must have for $a = a + 1$ to make sense, and, as noted by Wirth [13], why $a = b$ doesn’t mean the same as $b = a$?

Inconsistencies are troubling: why, along with full words like “static”, abbreviations such as “args” and “println”? Students will retain from that first exposure to programming that it’s not necessary to be consistent, and that saving keystrokes is more important than choosing clear names. (The feature that goes to the next line in the basic I/O Eiffel library is called *put_new_line*.) If indeed at some later stage we introduce programming methodology advice about choosing clear and consistent names, we can hardly expect the students to take us seriously. “Do as I say, not as I do” is another example of a dubious pedagogical technique.

To cite another example, when describing the need for a mechanism for treating operations as objects, like Eiffel’s agents or the closures of other languages, I had to explain how one addresses the issue in a language such as Java that doesn’t have these mechanisms. Since I used iterators as one of the motivating examples I was at first happy to find that the Sun page describing Java’s “inner classes” also has code for an iterator design, which it would have been nice to use as a model. But then it includes declarations such as

→ Chapter 19.

java.sun.com/docs/books/tutorial/java/jav-aOO/innerclasses.html, as of September 2006.

```
public StepThrough stepThrough() {  
    return new StepThrough();  
}
```

I can perhaps try to justify this to seasoned programmers, but there is no way I can explain it to someone who is just beginning — and I admire anyone who can. Why does `StepThrough` appear three times? Does it denote the same thing each time? What does the whole thing mean anyway? Very quickly the introductory programming course turns into exegesis of the programming language, with little time left for real concepts. In Alan Perlis’s words, “A programming language is low-level when its programs require attention to the irrelevant”.

Epigram #8, available at www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html as of September 2006.

Also contributing to the difficulties of using Java in an introductory course are the liberties that the language takes with object-oriented principles. For example:

- If x denotes an object and a one of the attributes of the corresponding class, you may by default write the assignment $x.a = v$ to assign a new value to the a field of the object. This violates information hiding and other design principles. To rule it out, you must shadow every attribute with a “getter” function, a tedious and unnecessary task. For the teacher, the choice is between forcing students early on to add considerable noise to their programs, or let them acquire bad design habits which are then difficult to unlearn.
- The Java notion of interface, separate from classes, forces a choice between fully abstract modules (interfaces) and fully implemented ones (classes). One of the benefits of the class mechanism, available as early as Simula 67, is to offer the full spectrum between these extremes. This is particularly useful to teach design: the first time you identify a notion you can express it through a fully deferred (abstract) class, which can then be refined through inheritance into a fully effective one. This is at the core of teaching the object-oriented method. The problem in Java is compounded by the inability to combine two or more abstractions through inheritance unless all but at most one are interfaces.

There are many more examples of such influences of Java on the teaching process; a new Eiffel user expressed a typical reaction by writing on a mailing list that *“I have written a lot of C++ and Java; all my brain power went on learning loads of nerdy computer stuff. With Eiffel I do not notice the programming and spend my time thinking about the problem.”*

A reason often invoked for using Java or C++ in introductory programming is the market demand for programmers in these languages. This is a valid concern, but it applies to the computer science curriculum as a whole, not to the first course. Programming at the level required of a CS graduate today is hard enough; we should use the best pedagogical tools. If market demand had been the determinant, we would never in the past have used Pascal (for many years the introductory language of choice), even less Scheme. Following the trends reflected in the latest ads for programmers we would in turn have imposed Fortran, Cobol, PL/I, Visual Basic, maybe C — and trained programmers who, a few years after graduation, would have found their skills obsolete when the great wheel of fashion turned. Our duty is to train problem-solvers who can quickly adapt to the evolutions of our discipline.

We should not let short-term market considerations damage pedagogical principles. In other words: if you think Java or C++ are ideal teaching tools, use them by all means; you probably won’t like this book very much. But if you agree with its approach, don’t be scared that some student or parent will complain that you use an “academic” approach. Explain to them that you are teaching programming in the best way you know, that someone who understands programming will retain that skill for life, and that any half-decent software engineer can pick up a new programming language at

breakfast — in case he or she hasn't already picked it up from other courses of your curriculum. As to the “academic” qualification (assuming that, in a university context, it is meant as derogatory!), point them to eiffel.com and its long list of mission-critical systems in Eiffel, successfully deployed by major companies, often after attempts in other languages had failed.

Java, C#, C++ and C are, for the next few years, an important part of any software engineer's baggage. Making sure the students know them is unrelated to the question of what techniques to use in the introductory course. Students will most likely be exposed to these languages at some point; it would be a rare curriculum these days where no course uses at least one of them. In any case no intro course that I know covers *all* of them, so students need to learn more regardless of the initial teaching language. This book goes further than standard textbooks in providing introductions to all of the languages cited.

Programming languages and the programming culture associated with each of them are interesting objects of study. My team at ETH, which teaches introductory programming in Eiffel, has introduced courses for the third year and beyond, each devoted to the detailed study of a specific language: “Java in Depth”, “C# in Depth” etc.

Students who know programming are well prepared to master the intricacies of specific languages. Eiffel is a benefit here too: as many people have noted, having learned Eiffel and its object model helps you become a better C++ or Java programmer.

As a potential employer in both academia and industry I see dozens of CVs every month. They all cite the same skills, including C++ and Java. Other than as checkboxes to be ticked, this will not impress anyone. What recruiters do watch for is any skill that sets out an applicant from the hordes of others with similar backgrounds. An example of such a distinctive advantage is that the applicant knows a fully object-oriented approach with support for software engineering, as evidenced by a curriculum using Eiffel and Design by Contract. It is possible to survive a C++-based curriculum without ever understanding O-O concepts in any depth; with Eiffel that's more unlikely. Competent employers know that what counts, beyond immediate skills, is depth of understanding of software issues and aptitude for long-term professional development. All the effort deployed through this book and the use of Eiffel is directed at these goals.

It may be appropriate here to cite Alan Perlis again: *A language that doesn't affect the way you think about programming is not worth knowing.* Epigram #19.

HOW FORMAL?

One of the benefits of the Design by Contract approach is to expose the students to a gentle dose of “formal” (mathematically-based) methods of software development.

The software world needs, among other advances, more use of formal methods. Any serious software curriculum should devote at least one course entirely to mathematics-based software development, based on a mathematical specification language. In addition — although not as a substitute for such a course — the ideas should influence the entire software curriculum, even if though (as discussed below) it is not desirable today to subject beginners to a fully formal approach. The challenge is not only to include an introduction to formal reasoning along with practical skills, but to present the two aspects as complementary, closely related, and both indispensable. The techniques of Design by Contract, tightly woven into the fabric of object-oriented program structures, permit this.

Teaching Design by Contract awakens students to the idea of mathematics-based software development. Almost from the first examples of interface specifications, routines possess preconditions and postconditions, and classes possess invariants. These concepts are introduced in the proper context, treated — as they should, although many programmers still fear them, and most programming languages offer no support for contracts — as the normal, obvious way to reason about programs. Without intimidating students with a heavy-duty formal approach, we open the way for the introduction of formal methods, which they will fully appreciate when they have acquired more experience with programming.

→ In chapter 4.

In no way does the use of a mathematical basis imply a stiff or intimidating manner. Some formality in the concepts goes well with a practical, hands-on approach. For example the text introduces loops as an *approximation* mechanism, to compute a solution on successively larger subsets of the data; in this view the notion of *loop invariant* comes naturally, at the very beginning of the discussion of loops, as a key property expressing the approximation obtained at every stage.

This emphasis on practicality distinguishes Design by Contract from the fully formal approaches used in some introductory courses, whose teachers hold that students should first learn programming as a mathematical discipline. Sometimes they go so far as to keep them away from the computer for a semester or a full year. The risk of such dogmatism is that it may produce the reverse of its intended effect.

Students, in particular those who have programmed before, realize that they can produce a program — not a perfect program, but a program — without a heavy mathematical apparatus; if you tell them that it’s not possible

they will just disconnect, and may from then on reject any formal technique as irrelevant, including simple ideas that can help them now and more advanced ones when they reach higher levels of expertise. As Leslie Lamport — not one to be suspected of lack of enthusiasm for methods — points out [6]:

[In American universities] there is a complete separation between mathematics and engineering. I know of one highly regarded American university in which students in their first programming course must prove the correctness of every tiny program they write. In their second programming course, mathematics is completely forgotten and they just learn how to write C programs. There is no attempt to apply what they learned in the first course to the writing of real programs.

My experience confirms this. First-year students, who react well to Design by Contract, are not ready for a fully formal approach. To develop a real appreciation for its benefits you must have encountered the difficulties of industrial software development. On the other hand, it also doesn't work to let students develop a totally informal approach first and, years later, suddenly reveal that there's more to programming than hacking. The appropriate technique, I believe, is incremental: introduce Design by Contract techniques right from the start, with the associated idea that programming is based on a mathematical style of reasoning, but without overwhelming students with concepts beyond their reach; let them master the practice of software development on the basis of this moderately formal approach; later in the curriculum, bring in courses on such topics as formal development and programming language semantics. This cycle can be repeated, as theory and practice reinforce each other.

Such an approach helps turn out students for whom correctness concerns are not an academic chimera but a natural, ever-present component of the software construction process.

OTHER APPROACHES

Looking around at university curricula, talking to teachers and examining textbooks leads to identifying four main approaches in use today for introductory programming:

- Language-focused.
- Functional (in the sense of functional programming).
- Formal.
- Structured, Pascal or Ada-style.

It is important to understand the benefits of these various styles — indeed we retain something from each of them — and their limitations.

The first approach is probably nowadays the most common. It focuses on a particular programming language, often Java or C++. This has the advantage of practicality, and of easily produced exercises (subject to the Google-and-Paste risk), but gives too much weight to the study of the chosen language, at the expense of fundamental conceptual skills. Relying on Eiffel helps us teach the concepts, not the specifics of a language.

The second approach is illustrated in particular by the famous MIT course based on the Scheme functional programming language [1], which has set the standard for high-level curricula. It is strong on teaching the logical reasoning skills essential to a programmer. We strive to retain these benefits, as well as the relationship to mathematics, present here through logic and Design by Contract. But in my opinion object technology provides students with a better grasp of the issues of program construction. Not only is an O-O approach in line with the practices of the modern software industry, which has shown little interest in functional programming; more importantly for our pedagogical goals, it emphasizes system building skills and software architecture, which should be at the center of computer science education.

While, as noted, the curriculum should not be a slave to the dominant technologies just because they are dominant, we have a duty of realism. Using techniques too far removed from practice subjects us to the previously mentioned risk of disconnecting from the students, especially the most advanced ones, if they see no connection between what they're being taught and what their incipient knowledge of the discipline tells them. (Alan Perlis put this less nicely: *Purely applicative languages are poorly applicable.*)

I would argue further that the operational, imperative aspects of software development, downplayed by functional programming, are not just an implementation nuisance but a fundamental component of the discipline of programming, without which many of the most difficult issues disappear. If this view is correct, we are not particularly helping students by protecting them from these aspects at the beginning of their education, presumably abandoning them to their own resources when they encounter them later.

It's useful to point out that O-O programming is as mathematically respectable — through the theory of abstract data types on which it rests and, in Eiffel, the reliance on contracts — and as full of intellectual challenges as any other approach. Recursion, one of the most fascinating tools of functional programming, receives extensive coverage in the present book.

→ Chapter 16.

Some of the comments on functional programming also apply to the next approach, reliance on formal methods. As argued above, a fully formal approach is, at the introductory programming level, premature. The practical effect may be to convince students that academic computer science has nothing to do with the practice of software engineering, and causing them to adopt a blasé, method-less approach to programming.

The fourth commonly used approach, pioneered at ETH, draws its roots in the structured programming work of the seventies, and is still widespread. It emphasizes program structure and systematic development, often top-down. The supporting programming language is typically Pascal, or one of its successors such as Modula-2, Oberon or Ada. The approach of this book is heir to that tradition, with object technology viewed as a natural extension of structured programming, and a focus on programming-in-the-large to meet the challenges of programming in the new century.

TOPICS COVERED

The book is divided into five parts.

Part **I** introduces the basics. It defines the building blocks of programs, from objects and classes to interfaces, control structures and assignment. It puts a particular emphasis on the notion of contract, teaching students to rely on abstract yet precise descriptions of the modules they use and, as a consequence, to apply the same care to defining the interface of the modules they will produce. A chapter on “Just Enough Logic” introduces the key elements of propositional calculus and predicate calculus, both essential for the rest of the discussion. Back to programming, subsequent chapters deal with object creation and the object structure; they emphasize the modeling power of objects and the need for our object models to reflect the structure of the external systems being modeled. Assignment is introduced, together with references, only after program structuring concepts. → *Chapter 5.*

Part **II**, entitled “How things work”, presents the internal perspective: basics of computer organization, programming languages, programming tools. It is an essential part of the abstraction-focused approach to make sure that students also master the *concrete* aspects of hardware and software, which define the context of system development. Programmers who focus on the low-level, machine-oriented, fine-control details are sometimes derided as “hackers” in the older sense (not the more recent one of computer vandal). There’s nothing wrong with that form of hacking when it’s the natural hands-on, details-oriented complement to the higher-level concepts of software architecture. Students must understand the constraints that computer technology puts on our imagination, especially orders of magnitude: how fast we can transmit data, how many objects we can store in primary and secondary memories, the ratio of access times for these two kinds.

Part **III** examines some of the fundamental “Algorithms and data structures” of computer science, from arrays and trees to sorting and some advanced examples. Here too the approach is object-oriented and library-based. It makes no attempt at

Part **IV** considers some more specialized object-oriented techniques such as inheritance, deferred features and constrained genericity, event-driven design, and a taste of concurrency.

Part **V** adds the final dimension, beyond mere programming, by introducing concepts of software engineering for large, long-term projects, with chapters on such topics as project management, requirements engineering and quality assurance.

Appendices provide an introduction to various programming languages of which the students should have a general understanding: C#, Java, C — described in some more detail since it's an important tool for accessing low-level details of the operating system and the hardware — and C++, a bridge between the C and O-O worlds.

Acknowledgments

A number of elements of this Instructor's Preface are taken from earlier publications: [\[7\]](#), [\[8\]](#), [\[9\]](#), [\[10\]](#), [\[12\]](#).

This book has its source in the “Introduction to Programming” course at ETH Zurich. One of the remarkable traditions of computer science at ETH is to entrust introductory courses to professors, unlike the practice of some institutions which view the teaching of introductory programming as a chore best handed over to part-timers or junior faculty. Being asked to teach the course was a great honor and *Touch of Class* is an attempt to repay my debt.

I have taught the course every Fall since 2003 and am greatly indebted to the outstanding assistant team that has built an extremely effective operation for handling exercise sessions, supporting students, devising exercises and exams, grading them, designing and supervising student projects, writing supplementary documents and teaching aids, even occasionally substituting for me in lectures. This has enabled me to concentrate on the core material, reassured that the logistics would work. I am also grateful to the hundreds of students who have now taken this course, put up with my trials and errors, and provided extremely useful feedback — not to mention outstanding software projects, the best kind feedback one can, in the end, hope for.

Many of them downloadable, both source and binary; see e.g. games.ethz.ch.

Bibliography

All URLs listed were correct as of June 2006.

- [1] Harold Abelson and Gerald Sussman: *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, 1996.
- [2] Bernard Cohen: *The Inverted Curriculum*, Report, National Economic Development Council, London, 1991.
- [3] Mark Guzdial and Elliot Soloway: *Teaching the Nintendo Generation to Program*, in *Communications of the ACM*, vol. 45, no. 4, April 2002, pages 17-21.
- [4] Joint Task Force on Computing Curricula: *Computing curricula 2001* (final report). December 2001, www.acm.org/sigcse/cc2001.
- [5] Joint Task Force for Computing Curricula 2005: *Computing Curricula 2005* (draft). April 4, 2005, www.acm.org/education/Draft_5-23-051.pdf.
- [6] Leslie Lamport: *The Future of Computing: Logic or Biology*; text of a talk given at Christian Albrechts University, Kiel on 11 July 2003, research.microsoft.com/users/lamport/pubs/future-of-computing.pdf.
- [7] Bertrand Meyer: *Towards an Object-Oriented Curriculum*, in *Journal of Object-Oriented Programming*, vol. 6, no. 2, May 1993, pages 76-81. Revised version in *TOOLS 11 (Technology of Object-Oriented Languages and Systems)*, eds. R. Ege, M. Singh and B. Meyer, Prentice Hall, Englewood Cliffs (N.J.), 1993, pages 585-594.
- [8] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997, especially chapter 29, "Teaching the Method".
- [9] Bertrand Meyer: *Software Engineering in the Academy*, in *Computer (IEEE)*, vol. 34, no. 5, May 2001, pages 28-35, se.ethz.ch/~meyer/publications/computer/academy.pdf.
- [10] Bertrand Meyer: *The Outside-In Method of Teaching Introductory Programming*, in Manfred Broy and Alexandre V. Zamulin, eds., Ershov Memorial Conference, volume 2890 of *Lecture Notes in Computer Science*, pages 66-78. Springer, 2003.
- [11] Christine Mingins, Jan Miller, Martin Dick, Margot Postema: *How We Teach Software Engineering*, in *Journal of Object-Oriented Programming (JOOP)*, vol. 11, no. 9, 1999, pages 64-66 and 74.
- [12] Michela Pedroni and Bertrand Meyer: *The Inverted Curriculum in Practice*, in *Proceedings of SIGCSE 2006* (Houston, 1-5 March 2006), ACM, se.ethz.ch/~meyer/publications/teaching/sigcse2006.pdf.
- [13] Niklaus Wirth: *Computer Science Education: The Road Not Taken*, opening address at ITiCSE conference, Aarhus, Denmark, June 2002, www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm.

Contents

Prefaces	v
student_preface	vii
Software everywhere	vii
Casual and professional software development	viii
Prior experience — or not	ix
Modern software technology	x
Object-oriented software construction	x
Formal methods	x
Learning by doing	xi
From the consumer to the producer	xii
Abstraction	xii
Destination: quality	xiii
instructor_preface	xv
The challenges of a first course	xv
Outside-In: the Inverted Curriculum	xix
The supporting software	xx
From programming to software engineering	xxii
Terminology	xxii
Technology choices	xxiii
Object technology	xxiii
Eiffel and Design by Contract	xxiv
Why not Java?	xxvi
HOW FORMAL?	xxx
OTHER Approaches	xxxi
Topics covered	xxxiii
Acknowledgments	xxxiv
Bibliography	xxxv

Contents	xxxvii
PART I: BASICS	5
1 The industry of pure ideas	5
1.1 THEIR MACHINES AND OURS	5
1.2 THE OVERALL SETUP	8
The tasks of computers	8
General organization	9
Information and data	10
Computers everywhere	11
The stored-program computer	12
1.3 KEY CONCEPTS LEARNED IN THIS CHAPTER	14
New vocabulary	15
1-E EXERCISES	15
2 Dealing with objects	17
2.1 A CLASS TEXT	17
2.2 OBJECTS AND CALLS	20
Editing the text	20
Running your first program	21
Dissecting the program	25
2.3 OBJECTS	28
Objects you can and can't kick	28
Features, commands and queries	29
Objects as machines	32
Objects: a definition	33
2.4 FEATURES WITH ARGUMENTS	34
2.5 KEY CONCEPTS LEARNED IN THIS CHAPTER	36
New vocabulary	36
2-E EXERCISES	36
3 Program structure basics	39
3.1 INSTRUCTIONS AND EXPRESSIONS	39
3.2 SYNTAX AND SEMANTICS	40
3.3 PROGRAMMING LANGUAGES, NATURAL LANGUAGES	41
3.4 GRAMMAR, CONSTRUCTS AND SPECIMENS	43
3.5 NESTING AND THE SYNTAX STRUCTURE	44

3.6	ABSTRACT SYNTAX TREES	45
3.7	TOKENS AND THE LEXICAL STRUCTURE	47
	Token categories	47
	Levels of language description	48
	Identifiers	48
	Breaks and indentation	49
3.8	KEY CONCEPTS LEARNED IN THIS CHAPTER	50
	New vocabulary	50
3-E	EXERCISES	50
4	The interface of a class	51
4.1	INTERFACES	51
4.2	CLASSES	53
4.3	USING A CLASS	55
	Defining what makes up a good class	55
	A mini-requirements document	56
	First class ideas	56
	What characterizes a metro line	57
4.4	QUERIES	58
	How long is this line?	58
	Experimenting with queries	59
	The stations of a line	61
	Properties of start and end lines	62
4.5	COMMANDS	63
	Building a line	63
4.6	CONTRACTS	65
	Preconditions	65
	Contracts for debugging	68
	Contracts for interface documentation	68
	Postconditions	68
	Class invariants	70
	Contracts: a definition	71
4.7	KEY CONCEPTS LEARNED IN THIS CHAPTER	72
4-E	EXERCISES	72
5	Just Enough Logic	73
5.1	BOOLEAN OPERATIONS	74

Boolean values, variables, operators and expressions	74
Negation	75
Disjunction	76
Conjunction	77
Complex expressions	78
Truth assignment	79
Tautologies	80
Equivalence	81
De Morgan's laws	83
Simplifying the notation	84
5.2 IMPLICATION	86
Definition	86
Relating to inference	87
Getting a practical feeling for implication	88
Reversing an implication	90
5.3 SEMISTRIC BOOLEAN OPERATORS	91
Semistrict implication	96
5.4 PREDICATE CALCULUS	97
Generalizing "or" and "and".	97
Precise definition: existentially quantified expression	99
Precise definition: universally quantified expression	100
The case of empty sets	101
5.5 FURTHER READING	103
5.6 KEY CONCEPTS LEARNED IN THIS CHAPTER	103
New vocabulary	104
5-E EXERCISES	104
6 Creating objects and executing systems	109
6.1 OVERALL SETUP	110
6.2 ENTITIES AND OBJECTS	111
6.3 VOID REFERENCES	113
The initial state of a reference	113
The trouble with void references	114
Not every declaration should create an object	115
The role of void references	117
6.4 CREATING SIMPLE OBJECTS	118

6.5	CREATION PROCEDURES	122
6.6	OBJECT CREATION: SUMMARY	126
6.7	CORRECTNESS OF A CREATION INSTRUCTION	126
6.8	SYSTEM EXECUTION	128
	Starting it all	128
	The root class, the system and the design process	129
	Specifying the root	130
	The current object and general relativity	130
6.9	KEY CONCEPTS LEARNED IN THIS CHAPTER	133
	New vocabulary	134
6-E	EXERCISES	134
	Talking about void	134
7	Control structures	137
7.1	PROBLEM-SOLVING STRUCTURES	137
7.2	THE NOTION OF ALGORITHM	139
	Example and definition	139
	Precision and explicitness: algorithms vs recipes	140
	Properties of an algorithm	141
	Algorithms vs programs	142
7.3	CONTROL STRUCTURE BASICS	144
7.4	SEQUENCE (COMPOUND INSTRUCTION)	145
	Examples	145
	Compound: syntax	147
	Compound: semantics	148
	Order overspecification	149
	Compound: correctness	150
7.5	LOOPS	151
	Loops as approximations	152
	The loop strategy	153
	Loop instruction: basic syntax	155
	Including the invariant	157
	Loop instruction: correctness	157
	Loop termination	159
	Animating a metro line	164
	Understanding and verifying the loop	168

The cursor and where it will go	171
7.6 CONDITIONAL INSTRUCTIONS	172
Conditional: an example	173
Conditional structure and variants	175
Conditional: syntax	179
Conditional: semantics	179
Conditional: correctness	180
7.7 THE LOWER LEVEL: BRANCHING INSTRUCTIONS	180
Conditional and unconditional branching	180
The goto instruction and flowcharts	181
Flowcharts	183
Goto harmful?	184
Structured programming	186
The goto puts on a mask	188
7.8 OTHER CONTROL STRUCTURES	189
Loop initialization	190
Other forms of loop	191
Multi-branch conditional	194
Preview: exception handling	194
7.9 TABLE-DRIVEN CONTROL	194
7.10 FURTHER READING	194
7.11 KEY CONCEPTS LEARNED IN THIS CHAPTER	195
New vocabulary	195
7-E EXERCISES	195
8 Routines and functional abstraction	197
8.1 BOTTOM-UP AND TOP-DOWN REASONING	197
8.2 ROUTINES AS FEATURES	199
8.3 ENCAPSULATING A FUNCTIONAL ABSTRACTION	200
8.4 ANATOMY OF A ROUTINE DECLARATION	201
Interface vs implementation	203
8.5 INFORMATION HIDING	204
8.6 PROCEDURES VS FUNCTIONS	205
8.7 FUNCTIONAL ABSTRACTION	206
8.8 USING ROUTINES	208

8.9 AN APPLICATION: PROVING THE UNDECIDABILITY OF THE HALTING PROBLEM	209
8.10 FURTHER READING	211
8.11 KEY CONCEPTS LEARNED IN THIS CHAPTER	211
New vocabulary	211
8-E EXERCISES	211
9 Variables, assignment and references	213
9.1 ASSIGNMENT	214
Summing travel times	214
Local variables	217
Function results	219
Entities and variables	220
Swapping two values	220
The power of assignment	220
9.2 ATTRIBUTES	223
Fields, features, queries, functions, attributes	223
Assigning to an attribute	224
Attributes and information hiding	225
9.3 KINDS OF FEATURE	228
The client's view	228
The supplier's view	231
9.4 ENTITIES REVISITED	232
Defining entities	232
Variable and constant attributes	232
9.5 REFERENCE ASSIGNMENT	234
9.6 CACHING AND THE SPACE-TIME TRADEOFF	238
9.7 KEY CONCEPTS LEARNED IN THIS CHAPTER	238
New vocabulary	238
Precise feature terminology	238
9-E EXERCISES	238
10 Fundamental data structures, genericity, and algorithm complexity	241
10.1 STATIC TYPING AND GENERICITY	241
Static typing	242
Static typing for container classes	242
Generic classes	243

Validity vs correctness	245
Classes vs types	246
Nesting generic derivations	248
10.2 CONTAINER OPERATIONS	249
Queries	249
Commands	250
Automatic resizing	251
Standardizing feature names for basic operations	252
10.3 ESTIMATING ALGORITHM COMPLEXITY	253
Measuring orders of magnitude	254
Mathematical basis	255
Making the best use of your lottery winnings	255
Abstract complexity in practice	256
Presenting data structures	256
10.4 ARRAYS	257
Bounds and indexes	258
Creating an array	259
Accessing and modifying array items	260
Bracket notation and assigner commands	261
Resizing an array	263
Using arrays	265
10.5 TUPLES	266
10.6 LISTS: LINKED, ARRAYED, MULTI-ARRAYED	268
Cursor queries	269
Cursor movement	272
Iterating on a list	273
Adding and removing elements	275
Linked lists	277
Programming with references	282
10.7 HASH TABLES	283
10.8 DISPENSERS	291
10.9 STACKS	293
Stack basics	294
Using stacks	294
Implementing stacks	297
10.10 QUEUES	301

10.11 OTHER STRUCTURES	304
10.12 ITERATING ON DATA STRUCTURES	304
10.13 FURTHER READING	306
10.14 KEY CONCEPTS LEARNED IN THIS CHAPTER	307
New vocabulary	308
10-E EXERCISES	308
11 Input, output and exceptions	309
11.1 READING FROM A FILE	309
11.2 WRITING OUT	309
11.3 ABNORMAL CASES: WHEN THE CONTRACT IS BROKEN	309
11.4 RECOVERING FROM EXCEPTIONS	309
PART II: HOW THINGS WORK	311
12 Just enough hardware	313
12.1 CODING DATA	314
The binary system	314
Binary basics	316
Basic representations and addresses	316
Powers of two	318
From cherries to bytes	319
“Real” numbers	319
12.2 MORE ON MEMORY	320
Persistence	320
Transient memory	321
Persistent memory	322
12.3 COMPUTER INSTRUCTIONS	323
12.4 GETTING A CONCRETE FEEL FOR COMPUTERS’ POWER	323
12.5 MOORE’S “LAW” AND THE EVOLUTION OF COMPUTERS	323
12.6 KEY CONCEPTS LEARNED IN THIS CHAPTER	323
New vocabulary	325
12-E EXERCISES	325
13 Describing syntax	327
13.1 THE ROLE OF BNF	327
Languages and their grammars	328
BNF basics	329

Distinguishing language from metalanguage	331
13.2 PRODUCTIONS	332
Concatenation	332
Choice	333
Repetition	333
Rules on grammars	335
13.3 USING BNF	337
Applications of BNF	337
Language generated by a grammar	337
Recursive grammars	338
13.4 DESCRIBING ABSTRACT SYNTAX	341
13.5 TURNING A GRAMMAR INTO A PARSER	342
13.6 THE LEXICAL LEVEL AND REGULAR AUTOMATA	343
Lexical constructs in BNF	343
Regular grammars	344
Finite automata	346
Context-free properties	348
13.7 FURTHER READING	349
13.8 KEY CONCEPTS LEARNED IN THIS CHAPTER	350
New vocabulary	350
13-E EXERCISES	351
14 Programming languages	353
15 Compilers and friends: the basic software tools	355
15.1 COMPILATION VS INTERPRETATION	355
15.2 ESSENTIALS OF A COMPILER	356
Lexical analysis, regular languages and finite automata	356
Parsing	356
Semantic analysis, code generation and optimization	356
What else a compiler does	356
15.3 LOADING, LINKING AND ALL THAT	356
15.4 TEXT AND PROGRAM EDITORS	356
15.5 CONFIGURATION MANAGEMENT	356
15.6 VIRTUAL MACHINES	356
15.7 SOFTWARE DEVELOPMENT ENVIRONMENTS	356

PART III: ALGORITHMS AND DATA STRUCTURES	357
16 Recursion and trees	359
16.1 BASIC EXAMPLES	360
Recursive definitions	360
Recursively defined grammars	361
Recursively defined data structures	361
Recursively defined algorithms	362
16.2 THE TOWER OF HANOI	365
16.3 BINARY TREES	370
A recursive routine on a recursive data structure	371
Children and parents	372
Recursive proofs	372
More binary tree properties and terminology	373
Binary tree operations	375
Traversals	375
Binary search trees	377
Performance	378
Inserting, searching, deleting	379
16.4 FROM LOOPS TO RECURSION	382
16.5 MAKING SENSE OF RECURSION	384
Vicious circle?	384
Boutique cases of recursion	386
Keeping definitions non-creative	388
The bottom-up view of recursive definitions	389
Bottom-up interpretation of a construct definition	392
The towers, bottom-up	393
Grammars as recursively defined functions	394
16.6 CONTRACTS FOR RECURSIVE ROUTINES	395
16.7 THE IMPLEMENTATION OF RECURSIVE ROUTINES	396
A recursive scheme	397
Routines and their execution instances	397
Preserving and restoring the context	398
Stacks	399
Taking advantage of invertible functions	403
16.8 GENERAL TREES	403
16.9 BACKTRACKING ALGORITHMS AND ALPHA-BETA SEARCH	403

16.10 APPENDIX: AVL TREES	403
16.11 APPENDIX: ITERATIVE HANOI	403
16.12 KEY CONCEPTS LEARNED IN THIS CHAPTER	405
New vocabulary	405
16.13 FURTHER READING	405
16-E EXERCISES	405
17 An elegant algorithm family: Topological Sort	409
17.1 THE PROBLEM	409
Examples	410
Points in a plane	411
17.2 THE BASIS FOR TOPOLOGICAL SORT	413
Binary relations	413
Acyclic relations	414
Order relations	415
Order relations vs acyclic relations	416
Total orders	418
Acyclic relations have a topological sort	420
17.3 PRACTICAL CONSIDERATIONS	421
Performance requirements	421
Class framework	422
Input and output	422
Overall form of the algorithm	423
Cycles in the constraints	424
Overall class organization	427
17.4 BASIC ALGORITHM	430
The loop	430
A “natural” choice of data structures	431
Performance analysis of the natural solution	432
Duplicating the information	433
Spicing up the class invariant	434
Numbering the elements	435
Basic operations	436
The candidates	437
The loop, final form	440
Initializations and their time performance	442
Putting everything together	445

17.5 LESSONS	446
Interpretation vs compilation	446
Time-space tradeoffs	448
Algorithms vs systems and components	448
17.6 KEY CONCEPTS LEARNED IN THIS CHAPTER	449
New vocabulary	449
17.7 TERMINOLOGY NOTE: ORDER RELATIONS	450
17-E EXERCISES	450
PART IV: OBJECT-ORIENTED TECHNIQUES	453
18 Inheritance	455
18.1 ENFORCING A TYPE: OBJECT TEST	455
Assignment attempt	455
19 Operations as objects: agents and lambda calculus	457
19.1 BEYOND THE DUALITY	457
19.2 WHY OBJECTIFY OPERATIONS?	459
Iteration, integration, observation, undoing	459
A world without agents	461
19.3 AGENTS FOR ITERATION	465
Basic iterating schemes	465
Iterating for predicate calculus	466
Agent types	467
The anatomy of an iterator	469
19.4 AGENTS FOR NUMERICAL PROGRAMMING	473
A numerical analysis note	474
19.5 OPEN OPERANDS	475
19.6 LAMBDA CALCULUS	479
Operations on functions	479
Lambda expressions	480
Currying	482
Generalized currying	484
Currying in practice	484
The calculus	485
Lambda calculus and agents	490
19.7 INLINE AGENTS	491
19.8 A PROBLEM WITH AGENT TYPES	493

19.9 OTHER LANGUAGE CONSTRUCTS	494
Agent-like mechanisms	495
Routines as arguments	496
Function pointers	496
Many-Little-Wrappers and nested classes	497
19.10 FURTHER READING	498
19.11 KEY CONCEPTS LEARNED IN THIS CHAPTER	498
New vocabulary	499
19-E EXERCISES	499
20 Event-driven design	501
20.1 EVENT-DRIVEN GUI PROGRAMMING	502
Good old input	502
Modern interfaces	502
20.2 TERMINOLOGY	504
Events, their types and their arguments	504
Keeping the distinction clear	508
Contexts	510
20.3 PUBLISH-SUBSCRIBE REQUIREMENTS	511
Publishers and subscribers	511
The model and the view	512
Model-View-Controller	514
20.4 THE OBSERVER PATTERN	515
About patterns	515
Observer basics	515
The publisher side	516
The subscriber side	518
Publishing an event	521
Assessing the Observer pattern	521
20.5 USING AGENTS: THE EVENT LIBRARY	523
20.6 SOFTWARE ARCHITECTURE LESSONS	527
Choosing the right abstractions	527
MVC revisited	527
Invest and profit	529
Assessing software architectures	529
20.7 FURTHER READING	530

20.8 KEY CONCEPTS LEARNED IN THIS CHAPTER	531
New vocabulary	532
20-E EXERCISES	532
21 Program correctness	535
21.1 DESIGN BY CONTRACT	535
21.2 HOARE TRIPLES	535
21.3 ASSIGNMENT RULES	535
21.4 RULES ON CONTROL STRUCTURES	535
21.5 AN EXAMPLE PROOF	535
PART V: TOWARDS SOFTWARE ENGINEERING	537
22 Overview of software engineering	539
22.1 BASIC DEFINITIONS	540
22.2 COMPONENTS OF QUALITY	542
Process and product	542
Immediate product quality	544
Long-term product quality	545
Process quality	547
Tradeoffs	549
22.3 MAJOR SOFTWARE DEVELOPMENT ACTIVITIES	549
22.4 LIFECYCLE MODELS	551
22.5 VERIFICATION AND VALIDATION	551
22.6 METRICS	551
22.7 CAPABILITY MATURITY MODELS	551
CMMI scope	552
CMMI disciplines	553
Goals, practices and process areas	553
Two models	554
Assessment levels	554
Goals and practices for each level	556
22.8 FURTHER READING	556
22.9 KEY CONCEPTS LEARNED IN THIS CHAPTER	557
New vocabulary	558
Acronym collection	558
22-E EXERCISES	558

Wrong or late?	559
23 The software process	561
24 Writing requirements and documentation	563
25 Designing Graphical User Interfaces	565
26 Testing and debugging	567
27 Towards software reuse	569
PART VI: APPENDICES	571
A Using the EiffelStudio environment	573
27.1 SHOWING A CLASS AND ITS QUERIES	575
[To be completed.]	576
B Eiffel syntax specification	577
C The C# language	579
New vocabulary	579
C-E EXERCISES	579
D The Java language	581
New vocabulary	581
D-E EXERCISES	581
E The C language	583
New vocabulary	583
E-E EXERCISES	583
F The C++ language	585
New vocabulary	585
F-E EXERCISES	585
Picture credits	587
Index	589
Change log	595
28.1 PSEUDOCODE AND TOP-DOWN DESIGN	601
Pseudocode	601
Subtrees	604
28.4 COMPUTING WITH OBJECTS	605
Cows	608

PART I:

Basics

We start with the essentials of programming: objects, classes, interfaces and contracts, and supporting concepts including logic and some elements about hardware.

1

The industry of pure ideas

1.1 THEIR MACHINES AND OURS

Engineers design and build machines. A car is a machine for traveling; an electronic circuit is a machine for transforming signals; a bridge is a machine for crossing a river. Programmers — “software engineers” — design and build machines too. We call our machines *programs* or *systems*.

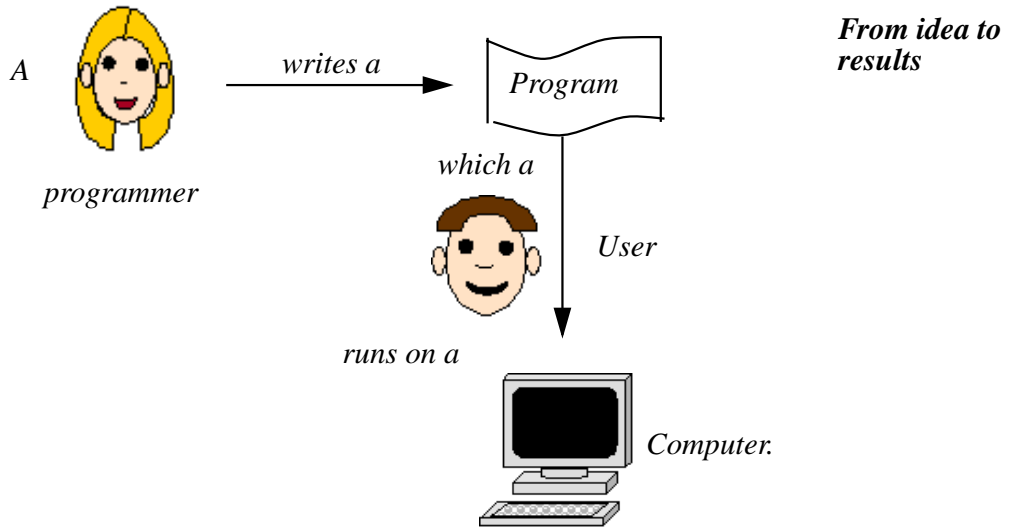
There’s a difference between our machines and theirs. If you drop one of their machines, it will hurt your feet. Ours don’t.

Programs are immaterial. This makes them closer, in some respects, to a mathematician’s theorems or a philosopher’s proposition than to an airplane or a vacuum cleaner. And yet, unlike theorems and propositions, they are engineering devices: you can operate a program, like you operate vacuum cleaners or planes, and get results.

Since one can’t operate a pure idea you will need some tangible, material support to operate programs or, using the more common terms, to *run* or *execute* them. That support is another machine: a **computer**. Computers and related devices are called **hardware**, indicating that — although they’re getting ever lighter — computers are the kind of machine that will hurt your feet. Programs and all that relates to them are by contrast called **software**, a word made up in the 1950s when programs emerged as topic of interest.

Here is how things work. You dream up a machine, big or small, and describe your dream in the form of a program. The program can then be fed into a computer for execution. The computer by itself is a general-purpose machine, but when equipped with your program it becomes a specialized machine, a material realization of the immaterial machine that you defined through your program.

The person who writes the program — “you” in the previous paragraph — is predictably called a *programmer*. Others, whom we call *users*, can then run your program on your computer, or theirs.



If you have ever used a computer, you’ve run some program, for example to browse the Web or play a DVD, so you’re already a user. This book should help you make it to the next step: programmer.

Cynics in the software industry pronounce “user” as “*loser*”. It’s one of the goals of this book that users of your programs will pronounce themselves winners.

The immaterial nature of the machines we build is part of what makes programming so fascinating. Given a powerful enough computer you can define any machine you want, whose operation will require billions upon billions of individual steps, and the computer will run it for you. You don’t need wood or clay or iron or anything that could wear you out carrying it up the stairs, burn you, or damage your clothes. State what you want, and you’ll have it. The only limit is your imagination.

Well, OK, that’s one of *two* limits; we don’t like to mention the other in genteel company, but you’ll likely encounter it before long. It’s your own fallibility. Nothing personal: if you are like the rest of us, you make mistakes. Lots of mistakes. In ordinary life they are not all harmful, as most human activities are remarkably error-tolerant. You can press your fork a little too intensely, drink your water a little too fast, push the accelerator a little too hard, use the wrong word; this happens all the time and in most cases doesn’t

prevent you from achieving what you wanted: eat, drink, drive, communicate. But programming is different! At a dazzling speed — hundreds of millions of basic operations per second — the computer will run your machine description, your program, exactly as you prepared it. The computer doesn't “understand” your program, it just runs it; the slightest mistake will be faithfully carried out by the machinery. What you wrote is what you get!

As you learn about programming in the following chapters, this is perhaps the most important property of computers to keep in mind. You might still believe otherwise: because computer programs do things that seem so sophisticated — like finding, in less than a second, your ideal vacation rental from millions of offers available on the World-Wide Web — we may easily succumb to the impression that computers are smart. Wrong. Although some programs may embody considerable human intelligence, the computer that runs them is like a devoted and unsufferable servant: infinitely faithful, almost infinitely fast, and definitely stupid. It will carry out your instructions exactly as you give them, never taking any initiative to correct mistakes, even those which a human being would find obvious and benign. The challenge for you, the programmer, is to feed this obedient brute with flawless instructions representing — in an execution of any significant program — billions of elementary operations.

If you've used computers you will know that they don't always react the way you like. It doesn't take very long to experience a “crash”, that state in which it seems everything goes away and execution stops. But, except for the infrequent case of a hardware malfunction, it's not the computer that crashed; it's a program that didn't do the right thing, and behind the program it's a programmer who didn't foresee all possible execution scenarios.

You can't learn programming without going through this experience of programs — yours, or someone else's — that do not work as they should; and you can't become a professional programmer without learning the techniques that will let you build programs that *do* work as you want.

The good news is that it is possible to produce such programs, provided you use the proper tools and maintain a lot of discipline, attention to the big picture as well as the details, and dedication.

1.2 THE OVERALL SETUP

In the next chapters we are going to jump right into program development. Initially we won't need too much detailed knowledge about computers, but let's see their fundamental properties, as they set the context for the construction of software.

The tasks of computers

Computers — “automatic stored-program digital computers” if we want to be precise — are machines that can store and retrieve information, perform operations on that information, and exchange information with other devices.

This definition highlights the major capabilities of computers:

What computers do

- **Storage and retrieval**
- **Operations**
- **Communication**

Storage and retrieval capabilities are a prerequisite for everything else: computers must be able to keep information somewhere before they can apply operations to it, or communicate it. Such a “somewhere” is called a **memory**.

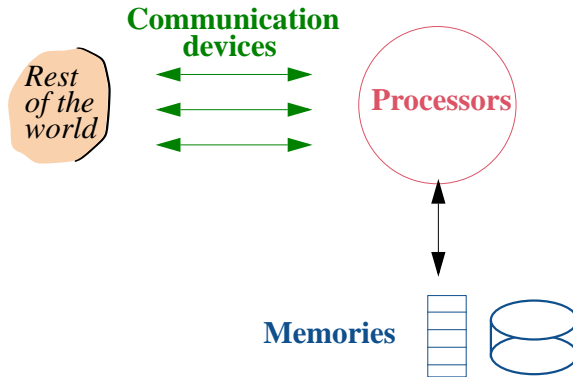
→ A more precise definition of “memory” appears below: page [12](#).

Operations include comparisons (“Are these two values the same?”), replacement (“Replace this value by that one”), arithmetic (“Find the sum of these two values”) and others. These operations are primitive; what makes computers able to perform amazing feats is not the intrinsic power of their basic mechanisms, but the *speed* at which they can carry them out and the ingenuity of the *humans* — that's you! — who write programs that will execute millions of them.

Communication allows us to enter information into computers, and retrieve information from them (the original information, or information that has been modified or produced by the computer's operations). It also enables computers to communicate with other computers and with devices such as sensors, telephones, displays and many others.

General organization

The previous definition yields the basic schematic diagram for computers:



*Components of
a computer
system*

The **memories** hold the information. We talk of memories in the plural because most computers have more than one storage device, of more than one kind, differing by size, speed of access to information and *persistence* (whether or not a memory retains information when power is switched off).

The **processors** perform the operations. Again there usually are several of them. Occasionally you'll still see a processor called a **CPU**, an acronym for the older term *Central Processing Unit*.

The **communication devices** provide means of interacting with the rest of the world. The figure shows the communication devices as interfacing with the processors rather than the memories; indeed, when exchanging information between a memory and the outside world, you will usually need to go through some operations of a processor. A communication device supports **input** (outside world to computer), **output** (the other way around), or sometimes both. Examples include:

- A keyboard, through which a person enters text (input).
- A video display or “terminal” (output).
- A mouse or joystick, enabling you to designate points on the terminal screen (input).
- A sensor, regularly sending measurements of temperature or humidity to a computer in a factory (input).
- A network connection to communicate with other computers and devices (input and output).

The abbreviation **I/O** covers both input and output. The words “input” and “output” are also used as verbs, as in “you must input this text”.

Information and data

The key word in the above definition of computers is “information”: what you would like to store into memories and retrieve from them, process through the processors’ operations, and exchange through the communication devices.

This is the human view. Strictly speaking, computers do not directly manipulate information, they manipulate *data* representing that information:

Definitions: Data, information

Collections of symbols held in a computer are called *data*.

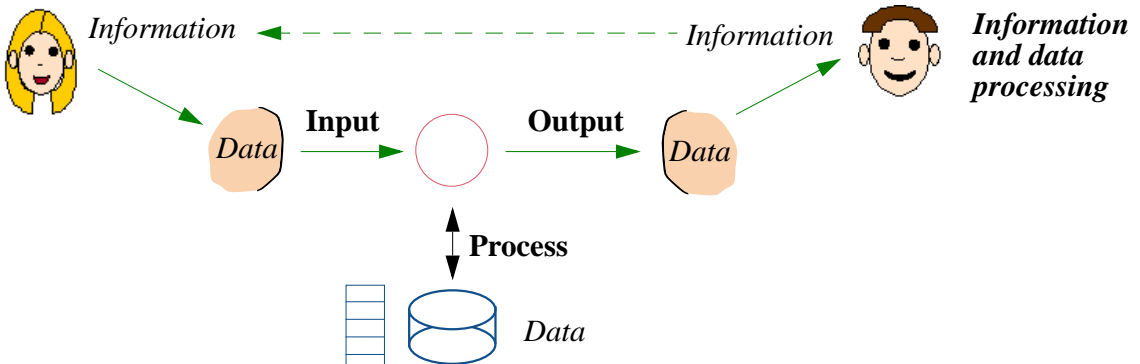
Any interpretation of data for human purposes is called *information*.

Some people will tell you that “data” should only be used in the plural, because it’s originally the plural of “*datum*”. Thank them for the kindness of their advice and disregard it cheerfully; unless they intend to continue the conversation in Latin, their linguistic data is obsolete.

Information is what you want: news from a Web site, a friend’s picture, background on someone you’ll be meeting. Data is how it’s encoded for the computer.

As an example, the MP3 audio format, which you may have used to listen to music with the help of a computer, is a way to encode enough *information* about a piece of music into *data* that can be stored in a computer, exchanged across a network, and sent to an audio device so that it will replay the music.

The data will be stored in memory. It’s the task of the communication devices to produce data from information coming from the world out there, store it in memory, and when the processors transform this data or produce new data, to send it out to the world so that it will understand it as information. In terms of the functions performed the previous picture looks like this:



The right-to-left arrow suggests that the process is not just one-way but repetitive, with information being repeatedly fed back to yield new results.

Computers everywhere

The familiar picture of a computer is the “desktop” or “laptop” computer, whose processor and memory components are hosted in a box of a size somewhere between a textbook like this one and a big dictionary; the terminal is often the biggest part. All this is at human size. At *hand* size we find the “PDA” (Personal Digital Assistant), supposedly useful for such tasks as calendar tracking, which are just computers with reduced human-interaction devices. At the higher end computers used for large scientific computations (physics, weather prediction...) can reach *room* size. This is of course nothing compared to computers of a generation ago, which took up *building* size for much less power.



*Computers:
desktop (a);
laptop (b); PDA
(c); processor to
be embedded
(d).*

Reduced to their central processor and memory components, computers can be much smaller than any of this. Increasingly, “the computer” is a device included — the technical term is **embedded** — in products or other devices. Today’s cars include dozens of small computers, controlling fuel delivery, braking, even windows. The printer connected to your desktop computer is not just a printing engine, it’s itself a computer, able to produce fonts, smooth images, restart on the next page after a paper jam. Electric razors include computers; *manual* razors might include one some day. (The more expensive razor *blades* already contain electronic tracking tags to fight theft.) Washing machines contain computers, and in the future *clothes* may have their own computers, helping to tune the washing process.

The computers you will use for exercises of this book are still of the keyboard-mouse-terminal-box kind, but keep in mind that software techniques have to cover a broader scope. Software for embedded systems must satisfy very high quality requirements: malfunctions in (for example) brake-control software can have terrible consequences, and you cannot fix them — as you would for a program running on your laptop — by stopping execution, correcting the error, and starting again.

The stored-program computer

A computer, as noted, is a universal machine: it can execute any program that you input into it.

For this input process you'll use communication devices, typically a keyboard and mouse. Text will appear on your terminal screen as you type it, seemingly as a direct result, but this is an illusion. The keyboard is an input device, the terminal a distinct output device; echoing the input text on the screen requires a special program, a *text editor*, to obtain this input, process it and display it. Thanks to the speed of computers, this usually happens fast enough to give the illusion of a direct keyboard-screen connection; but if the computer responds more slowly, perhaps because it's running many programs at the same time, you may notice a delay between typing characters and seeing them displayed.

When you input the program, where does it go? *Memories* are available to host it. That's why we talk of *stored-program* computers: to become a specific machine ready to carry out the specific tasks that you (as the programmer) have assigned to it, the computer will read its orders from its own memory.

This property of computers explains why we haven't seen a proper definition of "memory" yet. We could have said that a memory is a device for storing and retrieving data; that's correct if we interpret the notion of stored-program computer to imply that programs are data. It's clearer, however, to mention programs explicitly:

Definition: Memory

A memory is a device for storing and retrieving data and programs.

This ability of computers to treat programs as data — *executable* data — explains their remarkable flexibility. At the dawn of the computer age, it led to visions of *self-modifying* programs (since a program can modify data, it can modify programs, including itself) and to some grandiose philosophizing about how programs were going, through successive self-modifications, to become ever more "intelligent" and take over the world. Closer to us, it's also the reason why email users are told to be careful about opening email attachments, since the data they contain could be a maliciously written program, whose execution will destroy other data.

For programmers, the stored-program property has a more immediate consequence: it makes programs amenable, like any other kinds of data, to various transformations through computer operations. In particular, the program you write is usually not the program you run. Codes that a processor can execute are designed for machines, not humans; using them directly to construct your programs would be tedious and error-prone. Instead you will:

- Write programs in notations designed for human consumption, called *programming languages*. This form of a program is called its **source** text (or source form, or just source).
- Rely on special programs called *compilers* to transform such human-readable program texts into a form (its target form) appropriate for processor execution.

→ Or “machine code”,
or “object form”, see ...

We’ll often encounter the following terms reflecting this division of tasks:

Definitions: Static, Dynamic

Static properties of a program are properties of its source text, which can be analyzed by a compiler. **Dynamic** properties are those characterizing its individual executions.

The details of all this — processor codes, programming languages, compilers, examples of static and dynamic properties — appear in later chapters. What matters for the moment is knowing that the programs you are going to write, starting with the next chapter, are meant not only executed by a computer (after suitable transformations) but also *understood by people*.

This human aspect of programming is central to the engineering of software. When you program you are not just talking to your computer but also to other people: whoever will be reading the program later, for example to add new functions or correct a mistake. That’s a good reason to worry about program readability; and it’s not just a matter of being nice to others, since that “whoever” might be *you*, a few months older, trying to decipher what in the world you had in mind when writing the original version.

Throughout this book we’ll emphasize, along with practices that make your programs good for the computer — for example, designing programs so that they will run fast enough — practices that make them good for your fellow programmers. Program texts should be understandable; programs should be *extendible* (easy to change); program elements should be *reusable*, so that when you’re faced later on with a similar problem you don’t have to reinvent the solution; programs should be *robust*, protecting themselves against unexpected input and abnormal circumstances; most importantly, they should be *correct*, producing the expected results.

Touch of history: It's all in the holes

Aerospace industry old-timers tell the story of the staff engineer who, in an early rocket project, was in charge of tracking the weight of everything that would get on board. He kept pestering the programmers about how much the control software would weigh. The reply, invariably, was that the software would weigh nothing at all; but he refused to accept it.

One day he came into the head programmers' office, waving a deck of punched cards (the input medium of the time, see the picture): "This is the software", he said, "Didn't I tell you it had a weight like everything else!". The chief programmer seized the deck of cards from him: "See these holes? They are the software."



***A deck of cards
ready to be
punched***

1.3 KEY CONCEPTS LEARNED IN THIS CHAPTER

- *Computers* are general-purpose machines. Providing a computer with a *program* turns it into a special-purpose machine.
- Computer programs process, store and communicate *data* representing *information* of interest to people.
- A computer consists of *processors*, *memories* and *communication devices*. These material devices together make up *hardware*.
- Programs and associated intellectual value are called *software*. Software is an engineering product of a purely intellectual nature.
- Programs must be stored in memory prior to execution. They may have several forms, some readable and intended for human use, others directly processable for execution by computers.

- Computers appear in many different guises; many are *embedded* in products and devices.
- Programs must be written to facilitate understanding, extension and reuse. They must be correct and robust.

New vocabulary

At the end of every chapter you'll find such a list. Check (this is the first exercise in the chapter) that you know the meaning of each term listed; if not, find its definition, as you'll need all terms in subsequent chapters. To find a definition, look up the Index, where definition pages appear in bold.

Communication device	Compiler	Computer
Correct	CPU	Data
Dynamic	Embedded	Extendible
Hardware	Information	Input
Output	Memory	Persistence
Processor	Programmer	Programming language
Reusable	Robust	Software
Source	Static	Target
Terminal	User	

1-E EXERCISES

1-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

1-E.2 Data and information

For each of the following statements, say whether it characterizes data, information or both (explain):

- 1 • “You can find the flight details on the Web.”
- 2 • “When typing into that field, use no more than 60 characters per line.”
- 3 • “Your password must be at least 6 characters long.”
- 4 • “We have no trace of your payment.”
- 5 • “You can’t really appreciate her site without the Macromedia Flash plug-in.”
- 6 • “It was nice to point me to your Web page, but I can’t read Italian!”
- 7 • “It was nice to point me to your Web page and I’d like to read the part in Russian, but my browser displays Cyrillic as garbage.”

1-E.3 Defining precisely something that you've always known

You know about alphabetical order: the order in which words are listed in a dictionary or other “*alphabetical*” list. Alphabetical order specifies, of two different words, which is “**before**” the other. For example the word *sofa* is before *soft*, which itself is before *software*.

The question you are asked in this exercise is simply:

Define under what exact conditions a word is alphabetically “before” another.

That is to say, define alphabetical order. This is a notion that you undoubtedly know to apply in practice, for example to look up your name in a list of candidates to an exam; what the exercise requests is a *precise* definition of this intuitive knowledge, of the kind you might need for a mathematical notion — or for a concept to be implemented in a program.

To construct your definition you may assume that:

- A **word** is a sequence of one or more letters. (It’s also OK to use “zero or more letters”, that is to say accept the possibility of empty words, if you find this more convenient. Say which version you are using.)
- A **letter** is one among a finite number of possibilities.
- The exact set of letters doesn’t matter but for any two letters it is known which one is “**smaller**” than the other. For example, with letters of the Roman alphabet, *a* is smaller than *b*, *b* is smaller than *c* and so on.

If you prefer a fully specified set of letters, just take it to include the twenty-six used in common English words, lower-case only, no accents or other diacritical marks: *abcdefghijklmnopqrstvwxyz*, each “smaller” than the next.

The problem calls for a definition, not a recipe. For example, an answer of the form “You first compare the first letters of the two words; if the first word’s first letter is smaller than the second word’s first letter then the first word is before the second, otherwise...” etc. is **not** acceptable since it is the beginning of a recipe, not a definition. A proper definition might start: “A word *w1* is before a word *w2* if and only if any of the following conditions holds: ...”.

Make sure that your definition covers all possible cases, and respects the intuitive properties of alphabetical ordering; for example it is not possible to have both *w1* before *w2* and *w2* before *w1*.

About this exercise: The purpose is to apply the kind of precise, non-operational reasoning essential in good software construction. The idea is borrowed from a comment of Edsger Dijkstra, a famous Dutch computer scientist.

2

Dealing with objects

You are now going to write and execute your first program and successive variants. You must be able to use the basic functions of a computer and find your way through its directories and files.

Also, EiffelStudio must be installed on your computer. Everything else you'll learn here.

In case something goes wrong at any time, remember this:

Touch of practice: If you mess up

It's possible, especially if you are not too experienced with computers, to make a mistake that will take you off the track carefully charted below. Try to avoid getting into that situation — that is to say, follow the instructions precisely — but if it happens don't panic; we've provided a recovery mechanism to let you restart on the right foot. Just go to [“RECOVERING FROM “FUBAR””, page 576](#).

2.1 A CLASS TEXT

Start EiffelStudio and open the project called *Traffic*. The precise details of how to do this are given in the EiffelStudio [appendix](#):

→ See [“SETTING UP THE PROJECT”, A.3, page 575](#).

Touch of practice: Using EiffelStudio

Since this book focuses on principles of software construction, the details of how to use the EiffelStudio tools to run the examples all appear in [appendix A: “Using the EiffelStudio environment”, page 573](#). To set up and run any example, turn to the corresponding section of that appendix. Be sure to read first its opening section, [“EIFFELSTUDIO BASICS”](#).

You will be looking at program texts both in this book and on your screen. You'll notice a few differences, since paper and screen have different requirements. In the book, you will note the following convention:

Touch of style:
Program text and explanation text

In this book, everything that's part of a program text appears in **this blue** (sometimes **bold** or *italics* according to precise conventions). Everything else is the book's explanations. So you should never confuse elements of the programs with observations about these programs.

Bring up the text of the "class" called *PREVIEW*. That text — once you've adapted it to your needs — will be the core of your first program. What you will see on the screen (with different font and color conventions) is:

→ See again: [A.3, page 575](#) on how to bring up the class.

```

class PREVIEW inherit
  TOURISM
feature
  explore is
    -- Show city info and route.
    do
    end
end
  
```

Declaration of the feature *explore*

The part you'll fill in

The first line says you are looking at a "class", one of the little immaterial machines out of which we build programs; it calls it *PREVIEW*, as indeed the class describes a small preview of a city tour.

The first two lines also state that *PREVIEW* will **inherit** from an existing class called (second line) *TOURISM*; this means that *PREVIEW* extends *TOURISM*, which already has lots of useful facilities, so all you have to do is add your own programming ideas in the new class *PREVIEW*. The class names reflect this relationship: *TOURISM* describes a general notion of touring the city; *PREVIEW* covers a particular kind of tour, not a real visit but a preview from the comfort of your desk.

Magic?

Class *TOURISM* is part of supporting software prepared specially for this book. By piggybacking on these predefined facilities, rather than building everything from scratch, you can learn the concepts one by one and immediately practice them by writing example programs.

So if it seems like magic that your first programs will work at all, it's not: the supporting software — the apparent “magic” — uses the same techniques that you will be learning throughout the book. Little by little we'll remove pieces of the magic, and at the end there won't be any left; you'll be able to reconstruct everything by yourself if you wish.

Even now, nothing prevents you from looking at the supporting software, for example class *TOURISM*; it's all in the open. Just don't expect to understand everything yet.

The text of a class describes a set of *features* or operations. Here there's only one, called *explore*. The part of the class that describes it is called the **declaration** of the feature. It consists of:

- The feature's name, here *explore*.
- The word **is**, a *keyword*.
- “-- Show city info and route.”, a *comment*.
- The actual content of the feature, enclosed in the keywords **do** and **end**, but empty for the moment: that's what you are going to fill in.

A **keyword** is a special word that has a reserved meaning; you may not use it for naming your own classes and features. To make keywords stand out we'll always show them in **bold** (blue, of course, since they're part of the color for program text). Here the keywords are **class**, **inherit**, **feature**, **is**, **do** and **end**. (With just these six you can already go quite a way!)

A **comment**, such as -- Show city info and route, is explanatory text that has no effect on the program execution but helps *people* understand the program text. Wherever you see “--” (two consecutive “minus” signs), it signals a comment, extending to the rest of the line. When you write a feature declaration you should **always**, as a matter of good style, include a comment after the **is** as here, to explain what the feature is about.

2.2 OBJECTS AND CALLS

Your first program will let you prepare a trip through a city that looks remarkably like Paris, which may be the reason why the program text calls it *Paris*. As this is your first trip let's play it safe. All we want our program to do is some display on the screen:

- First, display a map of Paris, including a map of the Metro (the underground train network).
- Next, spotlight, on the map, the position of the Louvre museum (you have heard about it, or maybe it's the only local name that you can pronounce at the moment).
- Next, highlight, on the Metro map, one of the metro lines — Line 8.
- Finally, since your ever thoughtful travel agent has prepared a route for your first trip through the city, animate that route by showing a picture of a little traveler hopping through the stops.

Editing the text

Programming time! Your first program

In this section you are asked to fill in your first program text, and then to run the program.

Here's what you should do. Edit the text of the class *PREVIEW* and modify the feature *explore* so that it reads like this:

explore is

-- Show some city info.

do

```
Paris.display
Louvre.spotlight
Line8.highlight
Route1.animate
```

end

The text you should type in

To avoid any confusion note the following about how to type in the text:

- The text of each line starts some distance away from the left margin; this is known as **indentation** and serves to show the structure of the text. As it has no effect on program execution, you could have everything left-aligned if you wanted to; but it has an effect on program understandability (and probably on your grade when you submit programs), so please observe it carefully. You’ll get general indentation rules in a later chapter.
- To achieve the indentation, don’t use repeated spaces, which could make it messy to align text; use the character marked *Tab* on your keyboard. Tabs automatically align to equally spaced positions.
- In *Paris.display* and similar notations on subsequent lines you see a period “.” between successive words. Unlike the period that terminates a sentence in written English, it doesn’t need to be followed by a space. Since it’s an important element it’s shown as a big blue dot, “.”, but on your keyboard it’s just the plain period character.
- More generally, the typographical variations — boldface, italics, color ... — don’t affect how you type the text, only how you *read* it, in this book and on the screen as displayed by EiffelStudio.

Running your first program

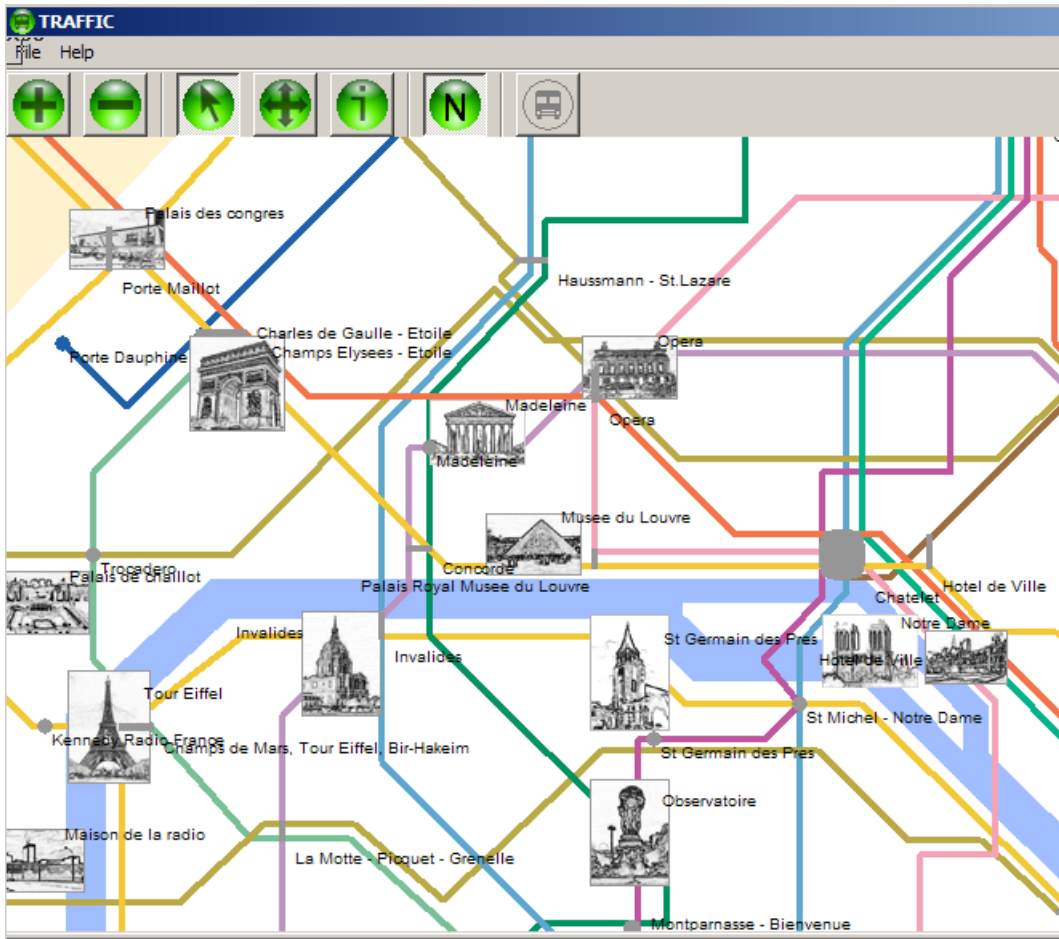
So much for the “cosmetics”, as programmers say — meaning superficial aspects of a program’s textual appearance.

You can run the program now by pressing the Compile and Run buttons of the EiffelStudio environment.

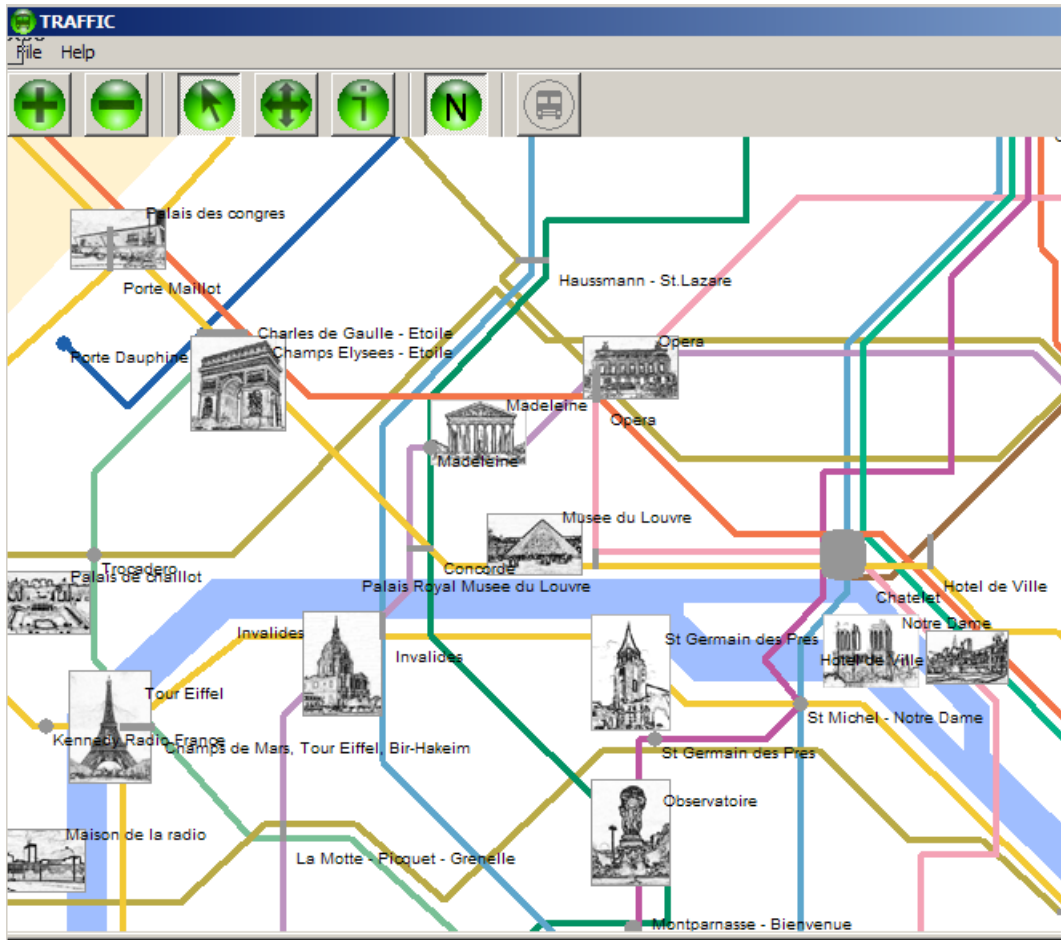
→ See details in
[“SETTING UP THE
PROJECT”, A.3, page
575.](#)

Do this now; you'll see the following sequence of events:

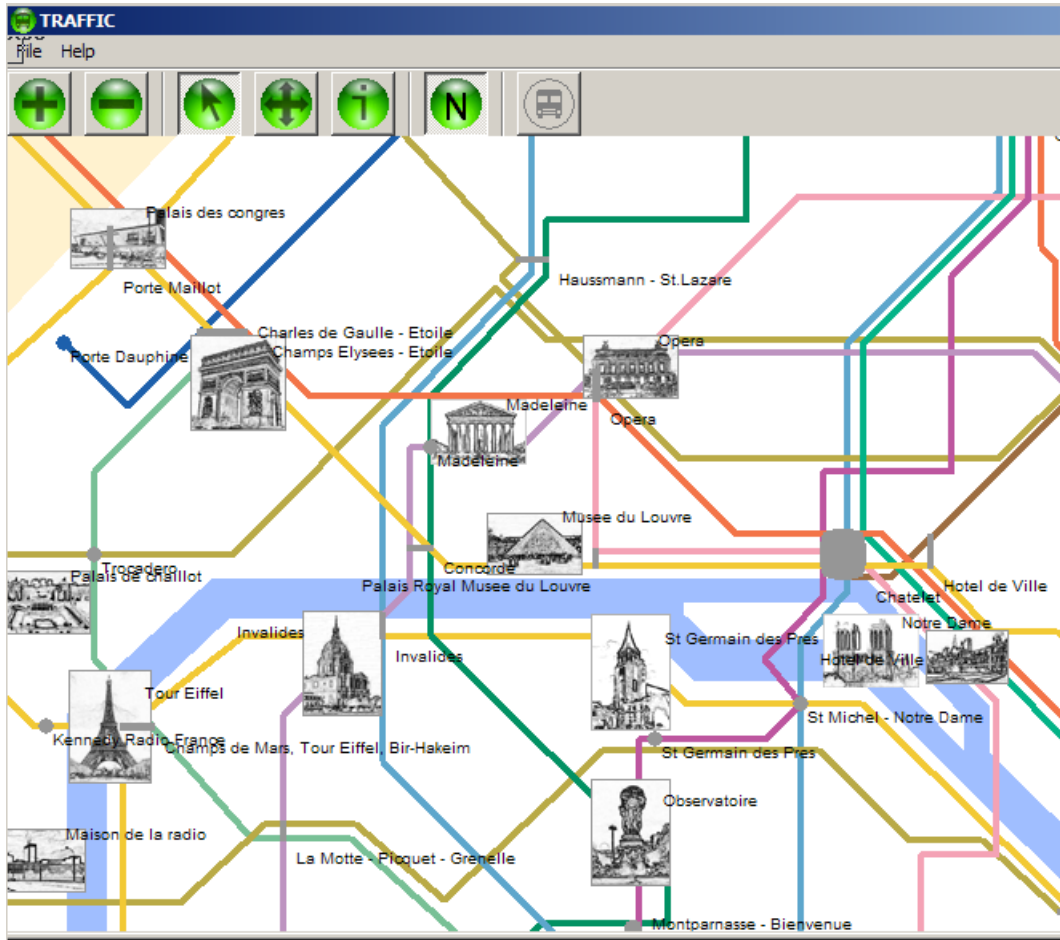
- 1 • As a result of executing the first line, *Paris.display*, the city map including the Metro network appears on the screen, like this:



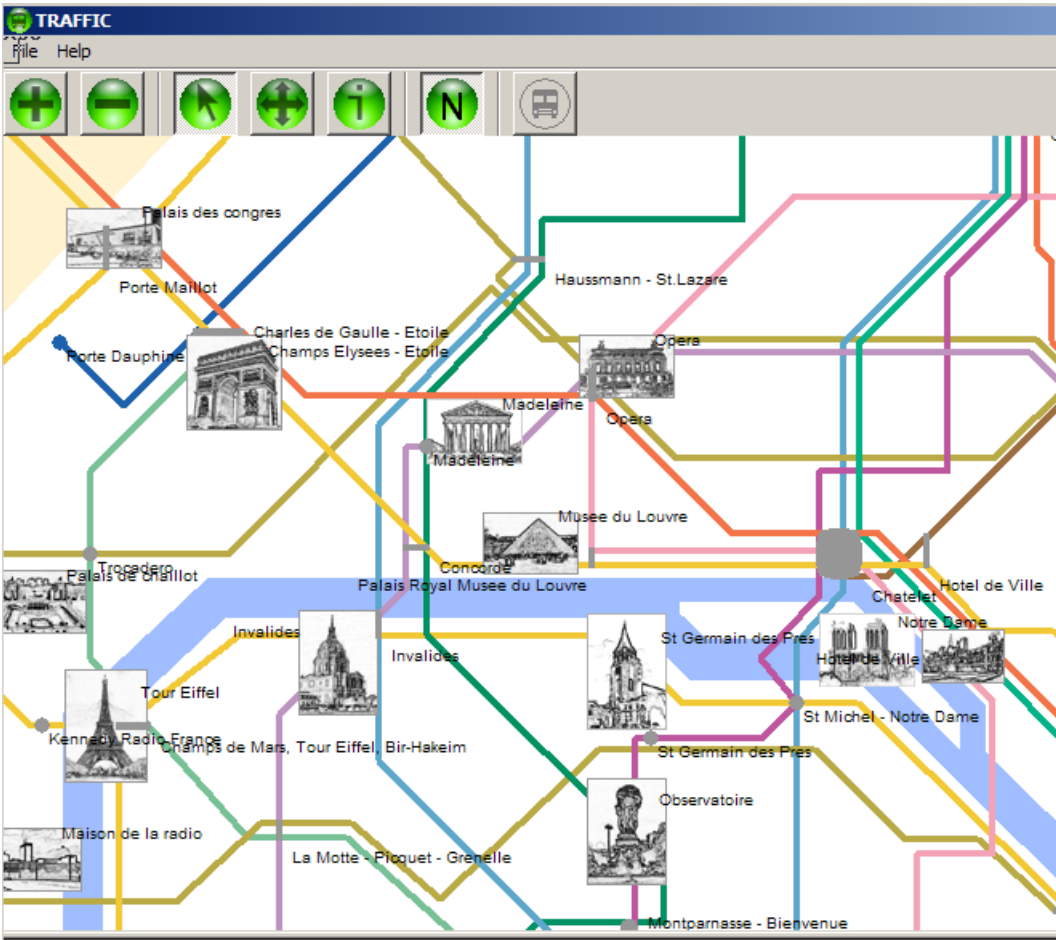
- 2 • Nothing happens for five seconds, then as a result of the second line *Louvre.spotlight* the position of the Louvre palace and museum shows up spotlighted on the map:



- 3 • After another five seconds, Line 8 of the Metro network comes up highlighted as a result of the third line *Line8.highlight*:



- 4 • After another short delay, the fourth line *Route1.animate* causes the plan to show a figurine representing a person and move it through the successive stops along the chosen route:



Once ready a program can, of course, be executed as many times as you like, so you can repeat the above process by pressing again the Run button.

Dissecting the program

The execution just described is the effect of the four lines that you inserted into the text of the feature *explore*. Let's look at what they mean. The techniques used in this simple program are fundamental; make sure that you understand everything in the following explanation.

The first line,

```
Paris.display
```

uses an object known in the program as *Paris* and applies to it the feature *display*. It's an example of a fundamental program operation:

$$x.f$$

where *x* denotes an object and *f* a feature (an operation). This simply means:

“Apply feature *f* to the object that *x* denotes”

This mechanism, known as **feature call**, applies a feature to an object. It is the very basis of computation: over and over again, that's what our programs do.

In our example the target object is called *Paris*. As the name suggests it represents a city. How much of the real city “Paris” does it really describe? You don't need to worry since *Paris* has been predefined for you. Pretty soon you will learn to define your own objects, but for the moment you have to rely on those we have set up for this exercise. Recognizing them is easy thanks to a basic convention:

Touch of style: Names of predefined objects

Names of predefined objects always start with an upper-case letter, as in *Paris*, *Louvre*, *Line8* and *Route1*. New names, corresponding to the objects that you define, will by default start with a lower-case letter.

Where are these “predefined” objects defined? You guessed it: in the class *TOURISM*, which your class *PREVIEW* inherits. This is where we put the “magic” through which your program, simple as it is, can produce significant results.

One of the features applicable to an object representing a city, such as *Paris*, is *display*, which graphically displays the current state of the city.

After applying *display* to the object *Paris*, the program performs another feature call:

$$\text{Louvre.spotlight}$$

The target object here is *Louvre*, another predefined object (name starting with a capital *L*) denoting the Louvre museum. The feature is *spotlight* which will spotlight the corresponding place on the map.

Then to highlight Line 8 we execute

```
Line8.highlight
```

using a feature *highlight* that highlights the target object, here *Line8* denoting an object that represents line number 8 of the underground system.

The final step, again a feature call, is

```
Route1.animate
```

where the target object is *Route1*, representing a predefined route — we assume, as noted, that it was all prepared by your travel agent — and the feature is *animate* which will showcase the route by moving a figurine along it.

For the program to work as expected, the features used in this program — *display*, *spotlight*, *highlight*, *animate* — must all do a little more than just displaying something on the screen. The reason is that computers are fast, *very* fast. So if the only effect of the first operation, *Paris.display*, was to display the map of Paris, the next operation, *Louvre.spotlight*, would follow a fraction of a second later; when you run the program you would never see the first display, the one that shows the map without the Louvre. To avoid this, the features all make sure, after displaying what they need to display, to pause execution for five seconds.

It's all taken care of in the text of these features, which we are not showing you yet (although you can look at them if you want to).

Congratulations! You have now written and run your first program, and you even understand what it does.

2.3 OBJECTS

Our example program works with *objects* — four of them, called *Paris*, *Louvre*, *Line8* and *Route1*. Working with objects is what all our programs will do; this notion of object is so fundamental that it gives its name to the whole style of programming: *Object-Oriented*, often abbreviated as “O-O”.

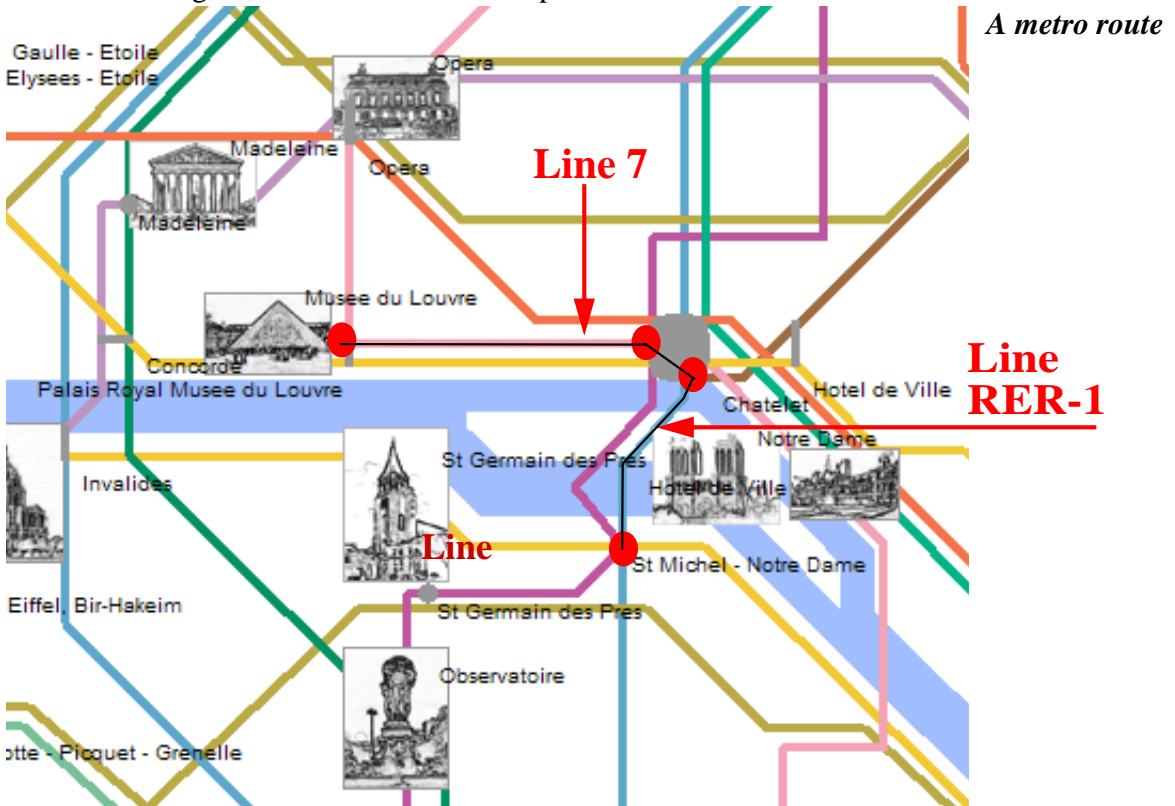
Objects you can and can't kick

What exactly should we understand from this word “object”? Here we are using for technical purposes a term from ordinary language — very ordinary language, since it's hard to think of a more general notion than “object”. Anyone can immediately relate to this word; that's both appealing and potentially confusing:

- It's appealing because using “objects” for your programs lets you organize them as models of real systems using real objects. If you do go to Paris you'll see that the Louvre is a real object; if its sight isn't enough to convince you of its reality, try kicking it with your foot. (Buying this book doesn't entitle you to a refund of medical expenses.) Our second software object so far, *Paris*, also corresponds to a real object, an even bigger one, the whole city.
- But this convenience of using software “objects” to represent physical ones should never lead you to confuse the two kinds. The reality of a software object doesn't extend beyond an immaterial collection of data stored in your computer; your program may have set it up so that operations on it represent operations on a physical object — like *Bus48.start*, representing the operation of making a bus move — but the connection is all in your mind. Even though our program uses an object called *Paris*, it's not the real Paris. (“You can't put Paris into a bottle”, says, more or less, an old French proverb, and you can't put Paris into a program either.)

Never forget that the word “object” as used in this book denotes a software notion. Some software objects represent things from the world out there, like the Louvre, but as we move to more sophisticated programming techniques that won't always be the case.

For example the last object we used, called *Route1*, represents a route — a travel plan. The particular plan represented by *Route1* enables you to go by Metro (underground) from the Louvre to Notre-Dame. As the bold black line shows on the figure, this route has three steps:



- Go from the “Louvre” station to “Châtelet” on line 7 (3 stops).
- Change lines
- Go from Châtelet to “Saint-Michel Notre-Dame” on line RER-1 (1 stop).

The “route” is this sequence of steps. It’s not a physical object that you can kick, like the Louvre or your little brother; but it’s an object all the same.

Features, commands and queries

What makes an object isn’t that it has a physical counterpart or not; it’s that we can manipulate it with our program through a set of well-defined operations, which we call **features**.

Some of the features applicable to a “route” object include *questions* that we may ask; for example:

- What is the starting point? What is the ending point? (For our example *Route1*, as described above: Louvre and Notre-Dame.)
- What kind of route is it: walking, by bus, by car, by metro, some combination of these? (For *Route1* the answer is: a metro route.)
- How many steps (legs) does it use? (For *Route1*: three of them, metro from Louvre to Châtelet, changing lines at Châtelet, metro from Châtelet to Notre-Dame.)
- What metro lines, if any, does it use? (For *Route1*: lines 7 and RER-1.)
- How many metro stations does it go through? (Here: four altogether.)

Such features, whose purpose is to obtain properties of an object, are called **queries**.

There's a second kind of feature, called a **command**; a command enables the program to change objects. We already used commands: in our first program, *Paris.display* changes the image displayed on the screen, so *display* is a command. In fact all four operations of our first program were commands.

As another set of examples, we may want to define the following commands on routes:

- Remove the first segment of the route, or the last segment, or any other specified by its index.
- Append (add at the end) a new segment; it must start at the current destination. For example we can append to *Route1* a new segment provided it starts at Notre-Dame, for example a metro segment from Notre-Dame to *Jussieu*, (4 stations, see map on the previous page); the route will be changed to involve 3 segments, 3 metro lines, and 8 stations; the result now starts at Louvre and ends at Jussieu.
- “Prepend” (add at the beginning) a new segment; it must end at the current origin. For example we can make *Route1* start with a segment going from *Opéra* to Louvre; this changes the number of stations but not the set of metro lines since Opéra is already on line 7.

All these operations change the route, and hence they are commands.

The tunnel signs that one encounters on a German *Autobahn* (freeway) are a good illustration of the command-query distinction. The sign at the entrance to a tunnel looks like this:



Command upon entering a tunnel

“*Licht !*”, you are told in no uncertain terms. Switch on your lights! Unmistakably a command.

When you exit the tone is more gentle:



Query upon leaving a tunnel

“*Licht ?*”: did you remember to switch off your lights? Just a query.

Objects as machines

The first thing we learned about programs is that they are machines. Like any complex machine, a program during its execution is made of many smaller machines. Our objects are those machines. ← *“The industry of pure ideas”, 1, page 5.*

Perhaps you find this hard to visualize: how can we see a travel route across the Metro as a *machine*? But in fact we just saw the answer: what characterizes a machine is the set of operations — commands and queries — that it provides to its users. Think of a DVD player, with commands to start playing, move to the next track, stop playing, and queries to display the number of the track being played, the time elapsed etc. To our programs, the *Route1* object is exactly like the DVD player: a machine with commands and queries.

The figure evokes this correspondence, with rectangular buttons representing commands and elliptic buttons queries.



*A “route”
object pictured
as a machine*

When thinking about objects — such as the one denoted by *Route1* — we now have two perspectives:

- 1 • The object covers a certain collection of data in memory, describing, in this case, all the information associated with a certain route — it has three segments, it starts at the station “Louvre” etc.
- 2 • The object is a machine, providing certain commands and queries.

These two views are not contradictory, but easy to reconcile: the operations that the machine provides (view 2) access and modify the data collected in the object (view 1).

Objects: a definition

Summarizing this discussion of objects, here is the precise definition that you'll have to remember throughout this book:

Definition: Object

An object is a software machine allowing programs to access and modify a collection of data.

In this definition and the rest of the discussion, to “access” data is to obtain the answer to a question about the data, without modifying it. (One could also say “consult” the data.)

It's good to have a precise definition too for the various kinds of operation that we apply to objects:

Definitions: Feature, Query, Command

An operation that programs may apply to an object is called a **feature**, and:

- A feature that *accesses* an object is called a **query**.
- A feature that may *modify* an object is called a **command**.

Examples of commands were *display* for a city such as *Paris*, *spotlight* for a monument or location such as *Louvre*. Queries have been mentioned, for example the starting point of a route, but we haven't actually used one yet.

Queries and commands work on existing objects. This means we'll need a third kind of operation: *creation* operations, to give us the objects in the first place. You don't have to worry about this for the moment because all the objects you need in this chapter — *Paris*, *Louvre*, *Route1* ... — are defined for you as part of the “magic” of class *TOURISM*, and at execution time they have already been created when your program needs to use them. Soon you'll learn to create your own objects as you please.

→ [Chapter 6](#).

This will also explain why (as you will remember if you read carefully) the notion of “machine” was used to characterize not only objects but also classes.

← *After the first version of class [PREVIEW](#) on page [18](#).*

2.4 FEATURES WITH ARGUMENTS

Queries are just as important as commands. Let’s see some examples of how to use them. We may take for example the starting point — the origin — of a route. It is given by a query *origin*, applicable to routes; its value for our example route *Route1* is written

```
Route1.origin
```

which is a feature call such as *Route1.animate* and the others we have seen, but in this case since the feature is a query the call doesn’t “do” anything; it simply yields a value, the origin of *Route1*. We could use this value in various ways, like printing it on a piece of paper; let’s instead display it on the screen.

You will have noticed at the bottom of the display a little window (rectangular area) marked “Console”; this is used to display information about the state of our city-modeling system. In our program it is — guess what — an object. You can manipulate it through the feature *Console*; it’s one of those predefined features, like *Paris* and *Route1*, that our example class *PREVIEW* “inherits” from *TOURISM*, our little “magic” class.

← See class *PREVIEW*, page 18.

One of the commands applicable to *Console* is called *show*; its effect is to display (show) a certain text in the console. Here we may use it to display the name of the starting point of the route.

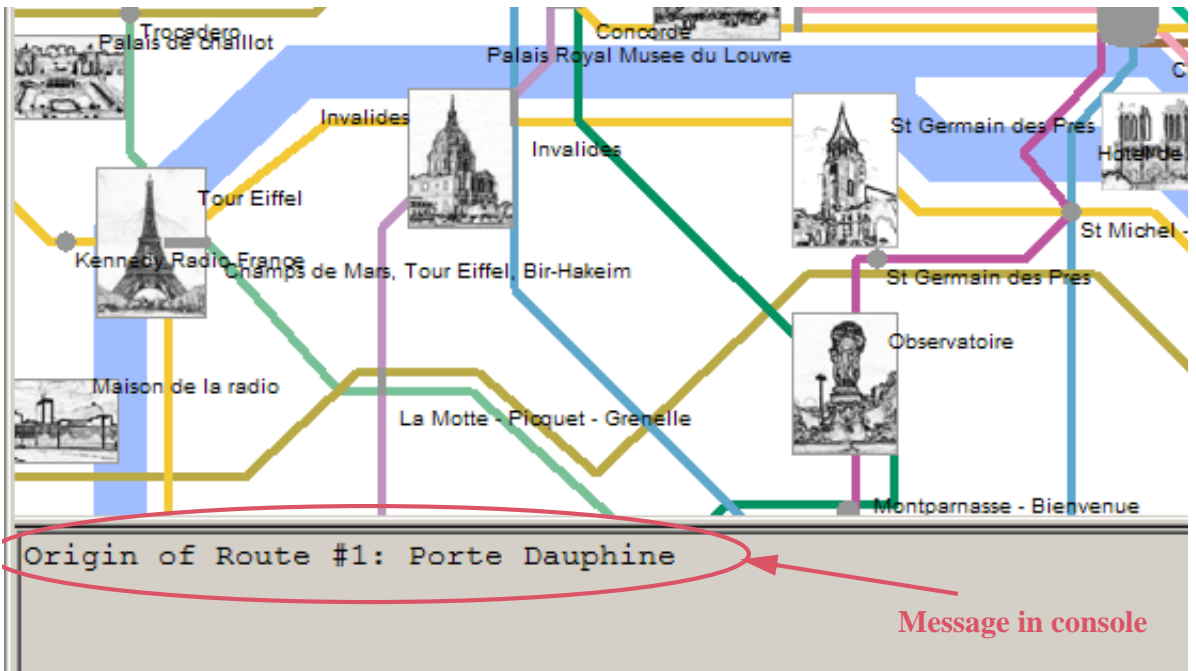
Programming time! **Displaying specific information**

You will now modify the previous program to make it display new information.

There are only two changes, as highlighted below: an update to the comment — for explanation purposes only — and a new operation at the end:

```
class PREVIEW inherit
  TOURISM
feature
  explore is
    -- Show city info, a route, and the route’s origin.
  do
    Paris.display
    Louvre.spotlight
    Line8.highlight
    Route1.animate
    Console.show (Route1.origin)
  end
end
```

Execute the resulting program; the origin of the route, *Louvre*, shows up in the console window:



This is the effect of our new feature call, *Console.show (Route1.origin)*. Previous feature calls were all of the form *some_object.some_feature*, but the form of this one is new:

```
some_object.some_feature (some_argument)
```

where *some_argument* is a value that we pass to the feature because it needs it to do its job. Feature *show*, for example, needs to know what to “show”, so we give it the corresponding value.

Such a value is known as an **argument** to the feature; the concept is the same as for arguments to functions in mathematics, where *cos (x)* denotes the cosine of *x* — the function *cos* applied to the argument *x*.

Some features will have more than one argument (separated by commas), but we won’t need them for a while; in well-designed software the vast majority of features typically have zero or one argument.

2.5 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A *program* is a set of mechanisms to create, access and change collections of information called *objects*.
- An object is a machine controlling a certain collection of data, providing the program, at run time, with a set of operations, called *features*, applicable to this data.
- Features are of two kinds: *queries*, which return information about an object; and *commands*, which can change the object.
- Some objects are software models of things from the physical world, like a building; others are software models of concepts from the physical world, like a travel route; others yet collect information that's meaningful to the software only.
- The basic operations performed by programs are *feature calls*, each of which applies a certain feature to a certain target object.
- A feature may have *arguments*, representing information it needs.

New vocabulary

Argument	Class	Command
Construct	Declaration	Feature
Feature call	Object	Query

“Class” awaits a more complete definition in the [chapter](#) on interfaces.

Chapter 4.

2-E EXERCISES

2-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

The definition of “class” may be less precise than the others.

2-E.2 Commands and queries

In software for creating, modifying and accessing documents, assume a class *WORD* that describes a notion of word, and a class *PARAGRAPH*, describing a notion of paragraph. For each of the following possible features of class *PARAGRAPH*, say whether it should be a command or a query:

- 1 • A feature *word_count*, used under the form *My_paragraph.word_count*, which gives the number of words in a paragraph.
- 2 • A feature *remove_last*, used under the form *My_paragraph.remove_last*, which removes the last word of a paragraph.

-
-
- 3 • A feature *justify*, used under the form *My_paragraph.justify*, which “justifies” a paragraph (makes sure it’s aligned to both the left and right margins, like the present paragraph and most others in this book, but not the margin notes such as the one adjacent to exercise [2-E.1](#)).
 - 4 • A feature *extend*, used under the form *My_paragraph.extend (My_word)*, which takes an argument representing a word and adds it at the end of the paragraph.
 - 5 • A feature *character_count*, used under the form *My_paragraph.character_count (i)*, which takes an integer argument representing the index of a word in a paragraph ($i = 1$ for the first word, $i = 2$ for the second word etc.) and gives the number of characters in the corresponding word (the word of index i) in the paragraph.

3

Program structure basics

The previous chapter gave us our first brush with programs. We are ready to move on to new concepts of software design; to make this experience more productive, let's pause for a moment and take a closer look at some of the program parts we have been using, so far without having names for them.

3.1 INSTRUCTIONS AND EXPRESSIONS

The basic operations that we instruct our computer to execute, like the five we had in the latest version

```
Paris.display  
Louvre.spotlight  
Line8.highlight  
Route1.animate  
Console.show (Route1.origin)
```

are, naturally enough, called **instructions**. It is customary to write just one instruction per line, as here, for program readability.

All of the instructions seen so far are feature calls. In subsequent chapters we will encounter other kinds.

To do its work, an instruction will need some **values**, in the same way that the mathematical function “cosine”, as in *cos (x)*, can only give you a result if it knows the value of *x*. For a feature call the needed values are:

- The target, an object, expressed as *Paris*, *Louvre* etc.
- The arguments, if any, such as *Route1.origin* in the last example.

Such program elements denoting values are called **expressions**. Apart from the forms illustrated here we will also encounter expressions of the standard mathematical forms, such as $a + b$.

Definitions: Instruction, Expression

In program texts:

- An **instruction** denotes a basic operation to be performed during the program's execution.
- An **expression** denotes a value used by an instruction for its execution.

3.2 SYNTAX AND SEMANTICS

In the above definitions of “instruction” and “expression” the word “denotes” is important. An expression such as *Routel.origin* or $a + b$ is not a value; it's a sequence of words in the program text. It *denotes* a value that will exist during the program's execution.

Similarly, an instruction such as *Paris.display* is a certain sequence of words, combined according to certain structural rules; it *denotes* a certain operation that will happen during execution.

This term *denotes* reflects the distinction between two complementary aspects of programs:

- The way you write a program, with certain words themselves made of certain characters typed on a keyboard: for example the instruction *Paris.display* consists of three parts, a word made of five characters *P, a, r, i, s*, then a period, then a word made of seven characters.
- The effect you expect the elements of these programs to have at execution: the feature call *Paris.display* will display a map on the screen.

The first kind of property characterizes the *syntax* of the programs, the second their *semantics*. Here are the precise definitions:

Definitions: Syntax, Semantics

The **syntax** of a program is the structure and form of its text.

The **semantics** of a program is the set of properties of its potential executions.

It's OK to use “semantics” as a singular, like other similar words: “*Economics was a big part of the minister's speech, but if the politics was obvious, the semantics was tortuous*”.

Since we write programs to execute them and obtain certain effects, it's the semantics that counts in the end, but without syntax there would be no program texts, hence no program execution and no semantics. So we'll need to devote our attention to both aspects.

Earlier on we had another distinction: *commands* versus *queries*. Commands are *prescriptive*: they instruct the computer, when executing the program, to do something for us, which may change objects. Queries are *descriptive*: they tell the computer to give the program some information about its objects, without changing these objects. Combining this distinction with the syntax-semantics division yields four cases:

	Syntax	Semantics
Prescriptive	Instruction	Command
Descriptive	Expression	Query Value

In the bottom-right entry we have two semantic concepts: a *query* is a program mechanism to obtain some information; that information itself, obtained by the program by executing queries, is made of *values*.

3.3 PROGRAMMING LANGUAGES, NATURAL LANGUAGES

The notation that defines the syntax and semantics of programs is a **programming language**. Various programming languages exist, serving different purposes; the one we use in this book is Eiffel.

Programming languages are artificial notations. Calling them “languages” suggests a comparison with the *natural* languages, like English or French, that we use for ordinary communication. Programming languages do share some characteristics with their natural cousins:

- The overall organization of a text as a sequence of *words* and *symbols*: a period “.” is a symbol; *PREVIEW* in Eiffel or “*The*” in English is a word.
- The distinction between *syntax*, defining the structure of texts, and *semantics*, defining their meaning.
- The availability both of words with a predefined meaning, such as “the” in English and **do** in Eiffel, and of ways to define your own words — as Lewis Carroll in *Alice in Wonderland*: “*Twas brillig, and the slithy toves...*”, and also as we just did by calling our first class *PREVIEW*, a name that means nothing special in Eiffel.

Word creation is far more common and open-ended with programming languages than with human ones. In English or French you don't invent new words all the time, unless you are a poet, or a little child, or maybe an Amazon botanist. In programming, people who've never seen a flower, even less one from South America, and outwardly appear adult, even possibly sane, might on a good day make up several dozen new names.

- Eiffel reinforces the human language flavor by drawing its keywords from English; every keyword of Eiffel is in fact a single and commonly used English word.

*With one exception: **elseif**, which agglomerates two English words.*

Some other programming languages use abbreviations, such as *int* for *INTEGER*, but we prefer full words for clarity.

- It's also recommended that, whenever possible, you use words from English or your own language for the names you define, as we did in the examples so far: *PREVIEW*, *display*, or *routel* (with a digit).

All these similarities between programming languages and human languages are good, because they help people understand programs. But they shouldn't fool you: programming languages are very different from human languages. They are *artificial* notations, designed for a specific purpose. This is both a loss and a gain:

- The power of expression of a programming language is ridiculously poor compared to the realm of possibilities available in any human language, even to a four-year old child. Programming languages can't express feelings or even thoughts: they define objects to be represented on a computer and tasks to be performed on these objects.
- What they miss in expressiveness, programming languages make up for in *precision*. Human texts are notorious for their ambiguity and openness to many interpretations, which are even part of their charm; when we tell computers what to do, we can't afford approximation. The syntax and semantics of a programming language must accordingly be defined very precisely.

Touch of Style: **Putting some English into your programs**

Natural language has a place in programs: in *comments*. We saw that any program text element that starts with two dashes "--" is, up to the end of the line, a comment. Unlike the rest of program texts, comments don't follow any precise rules of syntax, but that's because they have no effect on execution — no semantics. They're just explanations, helping people understand your programs.

In the end, to call our notations “languages” is to do them a favor they don’t quite deserve. Rather than scaled-down versions of the languages that people use to address each other, they are slightly scaled-up versions of the mathematical notations that scientists and engineers use to express formulas.

The term **code**, meaning “text in some programming language”, reflects this. It’s used in the expression “Line of code”, as in “*Windows XP contains more than 40 million lines of code*”. It’s also used as a verb: “to code” means to program, often with an emphasis on the lower-level aspects rather than the design effort, as in “*they think all the ideas are there and all there remains to do is coding*”. “Coder” is a somewhat derogatory term for “programmer”.

Still, programming languages have a beauty of their own, which I hope you will learn to appreciate. When you start thinking of your love life as *relationship.is_durable*, or sending your mom an SMS that reads *Me.account.wire(month.allowance + (month+1).allowance + 1500, Immediately)*, it will be a sign that either or both: (1) the concepts are starting to seep in; (2) it’s time to put this book aside and take the week-end off.

3.4 GRAMMAR, CONSTRUCTS AND SPECIMENS

To describe the syntax of a human language — meaning, as we have just seen, the structure of the corresponding texts — a linguist will propose a *grammar* for that language. For the simple English sentence

Isabelle calls friends

a typical grammar would tell us that this is a case (we’ll say a *specimen*) of a certain “construct”, maybe called “simple verbal sentence” in the grammar, with three component, each a specimen of some construct:

- The subject of the action: *Isabelle*, a specimen of the construct “**Noun**”.
- The action described by the sentence: *calls*, a specimen of the construct “**Verb**”.
- The object of the action: *friends*, another specimen of **Noun**.

Exactly the same concepts will apply to the syntax description of programming languages. For example:

- A *construct* of the Eiffel grammar is **Class**, describing all the class texts that anyone can possibly write.
- A particular class text, such as the text of class *PREVIEW* or class *TOURISM*, is a *specimen* of the construct **Class**.

A future chapter discusses in detail how to describe syntax, so for the moment we only need the basic definitions: → Chapter [13](#).

Definitions: Grammar, Construct, Specimen

A **grammar** for a programming language is a description of its syntax.

A **construct** is an element of a grammar describing a certain category of possible syntax elements in the corresponding language.

A **specimen** of a construct is a syntactical element.

→ “Grammar” will have a more detailed definition on page [328](#). A justification for using the term “specimen” appears on page [332](#).

Be sure to note the relationship between constructs and specimens. A construct is a type of syntactical element; a specimen is an instance — a specific example — of that type. So:

- In a grammar for English, we may have the constructs **Noun** and **Verb**; then *Elizabeth* is a specimen of **Noun**, and *rabbits* is a specimen of **Noun**.
- The standard grammar of Eiffel has the constructs **Class** and **Feature**; a particular class text is a specimen of **Class**, and any particular feature text is a specimen of **Feature**.

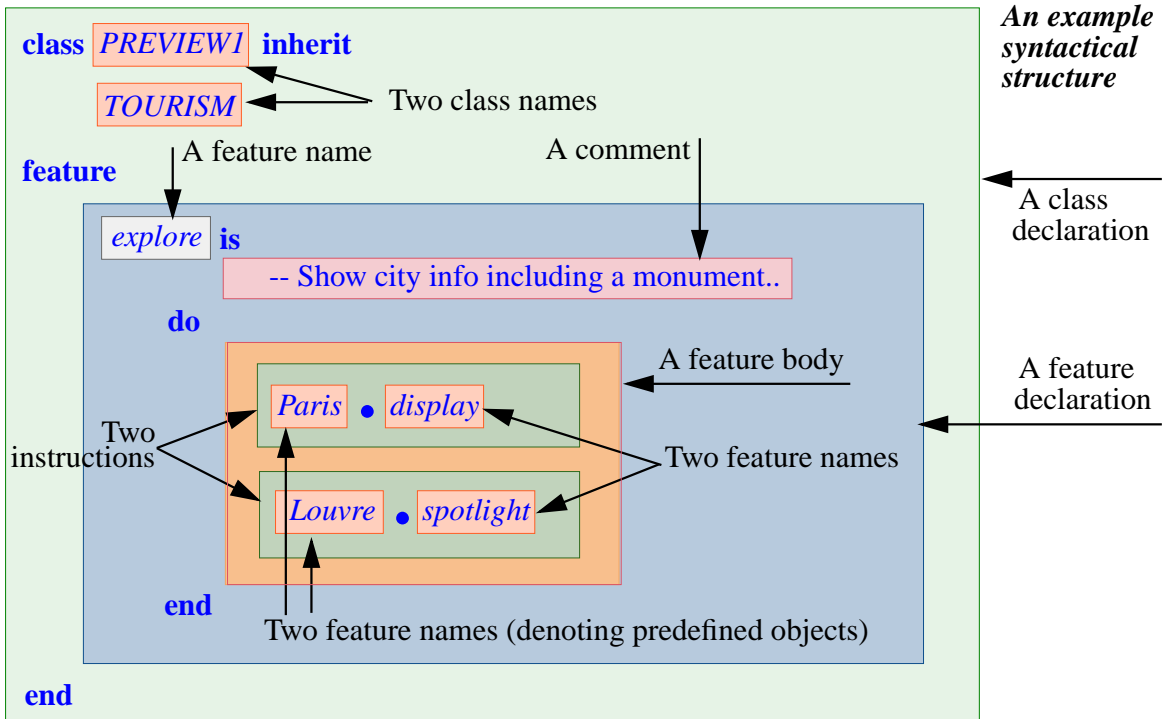
As these examples indicate, construct names will always appear in **This_green**, with an upper-case first letter. They are not program elements, but ways to *describe* certain categories of program elements, for example classes and features. Specimens are program elements, and so will appear, like all program text, in *this_blue*.

3.5 NESTING AND THE SYNTAX STRUCTURE

The syntax structure of a software text can involve several levels of specimens (syntactic elements). A class is a specimen; so is an instruction, or a feature name like *display*.

As these examples indicate, specimens can be embedded within other specimens or, as the technical term goes, **nested**.

Here is the nesting structure of specimens in our example class (retaining only two instructions for simplicity, and with a new name *PREVIEW1* to distinguish it from the full version):



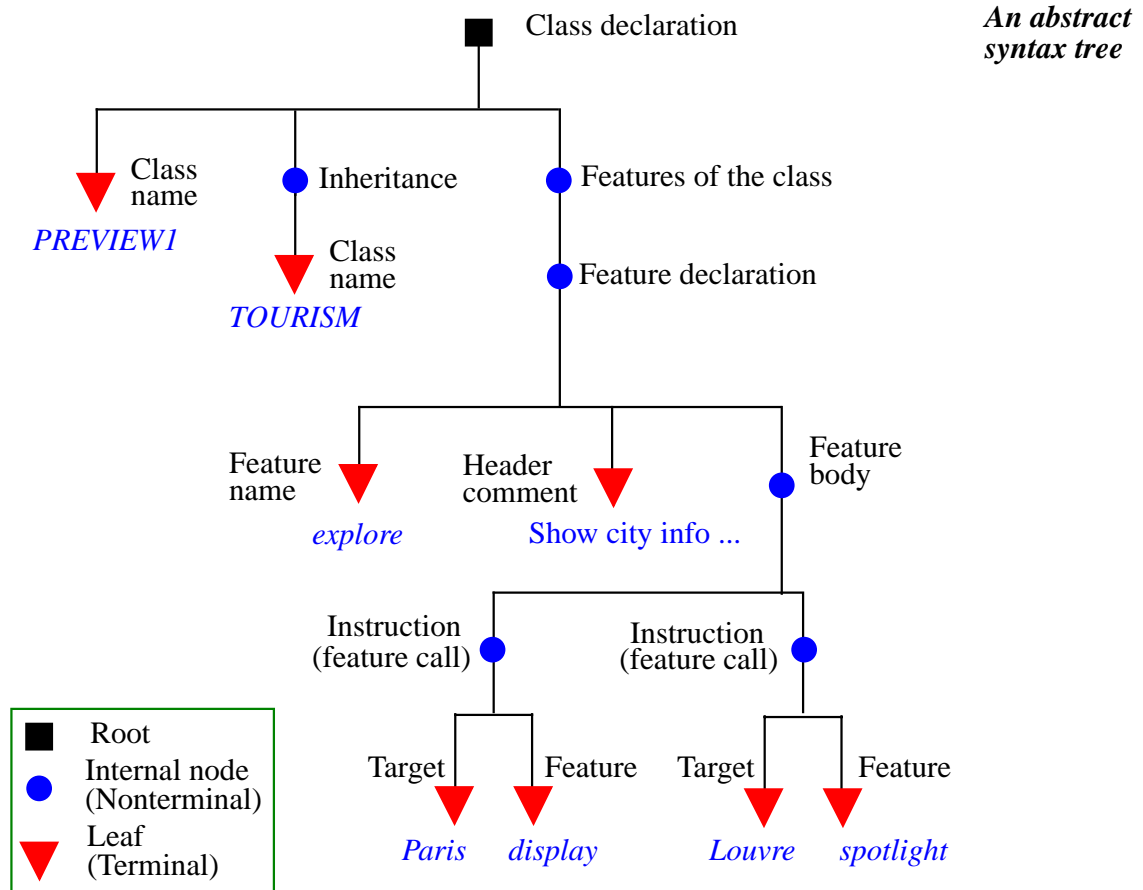
The embedding of the colored rectangles highlights the nesting of the specimens: the outermost rectangle covers the class declaration; it contains, among other specimens, a feature declaration; the feature declaration contains a “feature body” (the part that appears between the keywords `do` and `end`); the feature body contains two instructions; and so on.

Some elements of the syntax — keywords like `class`, `do`, `end`, and the period in feature calls — serve purely as delimiters and do not carry any semantic value of their own. We don’t consider them specimens.

Make sure you understand the syntactical structure as illustrated above.

3.6 ABSTRACT SYNTAX TREES

For larger program texts, another representation of such a structure is more convenient. It relies on the notion of **tree**, as used for example to represent the organizational chart of a company — and inspired from nature’s own trees with their branches and leaves, although *our* trees tend to grow top-down or left-to-right. A tree has a “root” which branches out to other “nodes” that may branch further. Trees serve to represent hierarchical structures as here:



This is known as an **abstract syntax tree**; it’s “abstract” because it doesn’t include the elements playing a delimiting role only, like the keywords **do** and **end**. We could also draw a “*concrete syntax tree*” that retains them.

A tree includes nodes and branches. Each branch connects a node to another. Any number of branches — including none at all — can leave out of any given node, but at most one branch may lead into it. A node with no incoming branch is a **root**; a node with no outgoing branch is a **leaf**; a node that is neither a root nor a leaf is an **internal node**.

A tree has exactly one root. (A structure made of zero, one or more disjoint trees, having any number of roots, is called a *forest*.) Trees are important structures of computer science and you will encounter them in many contexts. Here we are looking at a tree representing the syntax structure of a program element, a class. It represents the nesting of specimens, with the three kinds of node:

- The root represents the overall structure — the outermost rectangle on the earlier figure. ← Page 45.
- Internal nodes represent substructures that contain further specimens; for example a feature call contains a target and a feature name. In the earlier figure, these were the rectangles containing other rectangles.
- Leaves represent specimens with no further nesting, such as the name of a feature or class.

For an abstract syntax tree, the leaves are also called **terminals**; a root or internal node is called a **nonterminal**.

Every specimen is of a specific kind: the topmost node represents a class; others represent a class name, an “inheritance” clause, a set of feature declarations etc. Each such kind of specimen is called a **construct**. The above syntax tree shows, for each node, the corresponding construct name. Depending on the specimens it represents, a construct is either a “terminal construct” or a “non-terminal construct”: the figure shows “Feature declaration” as a non-terminal and “Feature name” as a terminal.

A construct is a general notion, for example the notion of a class; a particular instance of that notion, such as a particular class, is a specimen of that construct. As another example, the particular feature call *Paris.display* is a specimen of the construct “feature call”.

The syntax of a programming language is defined by a set of constructs and the structure of these constructs.

3.7 TOKENS AND THE LEXICAL STRUCTURE

The basic constituents of the syntax structure include terminals, keywords, and special symbols such as the period “.” of feature calls. These basic elements are called **tokens**.

Tokens are similar to the words and symbols of ordinary languages. For example the sentence in the margin has nine words (“This”, “is” etc.) and three symbols (two hyphens and the final period)

This is a nine-word and three-symbol sentence.

Token categories

We may divide tokens into two kinds:

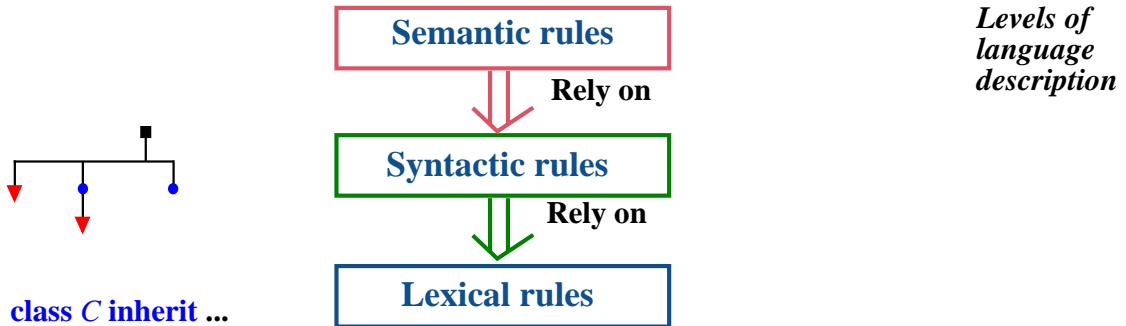
- **Terminals** correspond, as we have seen, to leaves of the abstract syntax tree; each carries some semantic information. They include names such as *Paris* or *display*, called **identifiers**, and chosen by each programmer to represent semantic elements such as objects (*Paris*) and features (*display*). Other examples are **operators** such as $+$ and \leq which will appear in expressions such as $a + b$, and **constants** denoting self-explanatory values, such as the integer 34.

- **Delimiters**, the second major category of tokens, do not directly carry any semantics but serve a purely syntactical role. They include the 65 or so **keywords** of the language, such as **class**, **inherit**, **feature**, and **special symbols** such as the period “.” of feature calls and the colon “:”.

Levels of language description

The form of tokens defines the **lexical** structure of the language. The syntax level comes above the lexical level, and semantics above syntax:

- Lexical rules define how to make up tokens out of characters.
- Syntax rules define how to make up specimens out of tokens satisfying the lexical rules.
- Semantic rules define the effect of programs satisfying the syntax rules.



Identifiers

For the moment we need only one lexical rule, governing identifiers:

Syntax: Identifiers

An identifier starts with a letter, followed by zero or more characters, each of which may be:

- A letter.
- A digit (0 to 9).
- An underscore character “_”.

Routel was an example of identifier including a digit.

You may define your own identifiers as you please based on this rule, except that you may not pick a keyword since it’s already reserved for a specific purpose. (Of course you don’t know all the keywords yet, but if you mistakenly reuse one of them you’ll get a clear error message.)

Touch of Style: Choosing your identifiers

For program readability, always choose identifiers that clearly identify the intended role; except in special cases (which we'll see), use full names, not abbreviations: *Route1*, not *R1* or *Rtel*.

There's no tax on keystrokes, and the fractions of seconds that you might save by omitting a letter will be more than offset by the time that you or someone else trying to understand your program will waste, later on, trying to figure out what you meant.

For identifiers denoting complex notions, use underscores to separate successive words, as in *My_route* or *bus_station*. This also works for class names, always in upper case: *PUBLIC_TRANSPORT*. Don't overdo it: for most identifiers, a single word, or two words separated by an underscore, are enough. Clear doesn't mean verbose.

In other programs you may encounter, for multi-word identifiers, the use of an upper-case letter in the middle of an identifier: *myRoute*, *PublicTransport*. Stay away from this convention, as it's far less readable than using underscores.

Breaks and indentation

The lexical structure consists of successive tokens. To separate adjacent tokens you may use a **break**, which is a sequence of one or more of the following:

- Space.
- *Tab* character (which shows up as a sequence of spaces to reach aligned positions, but internally is just one character).
- Return to the next line.

Breaks only serve to separate tokens. It makes no difference to the syntax and semantics whether you go to the next line or use one space, one or more tabs (typically at the beginning of a line, for indentation), or several spaces (seldom useful). This is what allows you to devise the text layout that will best reflect the program's structure — especially by highlighting its syntax nesting — to help readability, as in the examples of this book.

Your program is stored in a file, which contains a sequence of characters such as letters, digits, tabs and symbols. In that file a return to the next line is — in most file formats used today — simply represented by a particular character, known as “*New Line*”. You will also encounter references to the “*Carriage Return*” character, a delightful reminder of the time when we typed our programs on typewriters; the print head was lodged in a little mechanical “*carriage*”, which at the end of a line we would “*return*” to the leftmost position to start typing the next line.

On the Windows operating system, a line return is actually encoded by two characters, a Carriage Return followed by a New Line.

A break is usually not required between an identifier and a symbol: you may write *a+b* without spaces, since this is not ambiguous. The style rules suggest including the break anyway for clarity: *a + b*.

3.8 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Programs are expressed in a *programming language*.
- A program has a *lexical structure*, defining the form of a program's basic elements (*tokens* separated by *breaks* such as spaces, tabs and line returns); a *syntax*, defining its hierarchical decomposition into elements (specimens) built out of tokens; and a *semantics* defining the execution-time effect of each specimen and of the whole program.
- The syntax structure usually involves nesting and may be described as a tree, known as an *abstract syntax tree*.

New vocabulary

Abstract syntax tree	Break	Carriage return
Code	Construct	Delimiter
Expression	Identifier	Indentation
Instruction	Internal node	Leaf
Lexical	Natural language	Nesting
New line	Node	Nonterminal
Operator	Root	Semantics
Special symbol	Specimen	Syntax
Terminal	Token	Tree
Value		

3-E EXERCISES

3-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

The definition of "class" may be less precise than the others.

3-E.2 Syntax and semantics

For each of the following statements, say whether it characterizes syntax, semantics, both, or neither (explain):

- 1 • "In a feature call, you must separate the target object from the feature name by a period."
- 2 • "In a feature call $x.f$, there's no need to put spaces before or after the period, although they wouldn't hurt."
- 3 • "Every feature call applies a feature to a certain object, the 'target' of the call."
- 4 • "If there is an argument, it must be in parentheses."
- 5 • "If you have two or more arguments, separate them by commas."

4

The interface of a class

In the previous chapters we have learned to build some software relying on existing elements. We are going to do more of this now by seeing how we can use previously written *classes*. This will also be an opportunity to gain new insights into this notion of class, fundamental to everything we do in programming, and to discover the concepts of *interface* and *contract*.

4.1 INTERFACES

Many of the key decisions about building and using software systems involve the notion of interface. We may define it in relation to the notions, useful on their own, of client and supplier:

Definitions: Client, Supplier, Interface

A **client** of a software mechanism is a system of any kind — such as a software element, a non-software system, or a human user — that uses it. For its clients, the mechanism is a **supplier**.

An **interface** of a set of software mechanisms is the description of techniques enabling clients to use these mechanisms.

Informally, then, an interface for a piece of software is the description of how the rest of the world may “talk to” the software.

The definition speaks of “An interface”, not “*The* interface”. There are indeed several kinds of interface, and a software element may offer more than one interface, of the same or different kinds. The two principal kinds are:

- A **user interface**, where the clients are people using a software system.
- A **program interface**, where clients are themselves software elements.

As an example of a *user interface*, consider a Web browser as shown (top part only) on the next page. Its user interface is the description of what people can do with the browser; it includes:



A user interface

- The specifications of the fields into which users may type their own texts, such as the Address field at the top.
- The properties of buttons (“Back” etc.) that users may click to obtain certain effects.
- The conventions for hyperlinks (left-click will lead to a new page, right-click opens a page in a different window, etc.).
- More generally the set of rules that govern the interaction between the browser and its users.

Such a user interface is *graphical*, meaning that it involves pictures and other two-dimensional elements. The computing profession, which has a crush on acronyms, calls this a **GUI** — Graphical User Interface — pronounced “*Gooey*”.

Other user interfaces involve no graphics but only text, as on older cell phones; they are called “text interfaces” or “command-line interfaces”.

If in previous dealings with computer systems you have encountered less-than-perfectly-friendly user interfaces, you'll probably agree that GUI design is an important part of software design. We'll learn some principles later in this book but for the moment what's more fundamental is the second kind of interface cited, *program interfaces*. Here too you have to learn a TLA (Three-Letter-Acronym): **API**, for "Abstract Program Interface" (the A is also understood as meaning "*Application*").

In the rest of this chapter we'll learn how APIs look for a particularly important kind of software element: the class. Since for the moment we are not concerned any more with user interfaces, we'll say indifferently "API", "program interface" or even just "interface" to mean exactly the same thing.

4.2 CLASSES

A previous discussion defined an object as "a software machine allowing programs to access and modify a collection of data". Such collections of data might represent, to stick to the examples we have seen: ← *Page 33.*

- A city, where the "access and modify" operations may include finding out about current traffic conditions and adding some vehicles to the traffic. We have used *Paris* as an example but of course we may have objects representing any other city provided we have the relevant information.
- A travel route. Again we may have many such routes, not just *Route1* as used in the original example.
- A list of cars waiting at a red light. Many possible objects again.
- Closer to the computer, an element of the GUI such as a button or a window on the screen. Of these too we'll have many.

There is a strong similarity between objects of every such "kind": the operations applicable to a city object such as *Paris* would also apply to other city objects, say *New_York* or *Tokyo*. They don't apply to a travel route object such as *Route1*, but operations applicable to *Route1*, such as adding a new segment to a route, would also apply to other routes.

What this tells us is that the objects our programs manipulate naturally classify themselves into certain kinds, or **classes**: the class of objects representing cities, the class of objects representing travel routes, the class of objects representing buttons on the screen...

"Class" is indeed the technical term. What characterizes objects of a given class is a common set of applicable operations — or *features* in the terminology introduced in the discussion of objects. Hence the definition:

Definition: Class

A class is the description of a set of possible run-time objects to which the same features are applicable.

In program texts, classes will stand out by always having names in all upper case, such as *CITY*, *ROUTE*, *CAR_LIST*, *WINDOW*. The names representing objects are in lower case, or, in the case of predefined objects such as *Paris*, with only the first letter in upper case.

A class represents a category of things; an object represents one of these things. The following terms express precisely this relationship between classes and objects:

Definitions: Instance, Generating class

If an object *O* is one of the objects described by a class *C*, then *O* is an **instance** of *C*, and *C* is the **generating class** of *O*.

CITY is a class, representing all possible cities (as we've decided to model them in our program); *Paris* denotes an object, an instance of that class.

This relationship between classes and objects is the usual one between a category and members of that category: "Human" is a category, "Socrates" is one of its members. If these were software notions we would say there's a class *HUMAN* and one of its instances is the object called *Socrates*.

In software the difference goes further:

- Classes exist only in the software text. As the definition of "class" says, a class is a *description*; it will be given by a **class text**, a software element describing the properties of the associated objects (instances of the class). In fact a program is just a collection of class texts.
- Objects — "collections of data" — exist only during the software's execution; you don't see them in the program text, although you will see there some names such as *Paris* and *Route1* denoting objects that will appear during execution.

As a consequence, the term "run-time object" appearing in the definition of "class" is redundant, since objects by definition exist only during program execution ("run time"). From now on we'll say just "object".

Finding appropriate classes is a central part of the task of software *design*, devoted to organizing the essential structure, or *architecture*, of a program — as opposed to writing down the details, or *implementation*.

4.3 USING A CLASS

You are now going to learn what a class looks like and how you can use it to build new classes — *client* classes — for your own programs.

The classes that we'll examine have been written to cover properties and operations relative to a metro network like the Paris metro. (Of course, for generality, everything should be tailorable to any other city.) Below for reference is part of the Metro plan.



Metro plan

Defining what makes up a good class

Assume we had to devise a software model for the Metro as seen by passengers. As in any software design problem, the key question will be: *What are the classes?* To find good classes answering this question, we search the problem domain for concepts that:

- Describe sets of objects (their future instances).
- Can be explained clearly.
- Can be characterized in terms of clearly defined **features**, including both *queries* and *commands*, applicable to the corresponding objects.

← “*Features, commands and queries*”, page 29.

A mini-requirements document

Can we find classes and their features to make up a software model of the Metro? Often a first step to design is simply to express in clear, simple language what the problem domain is about. Let's try:

Touch of Paris: Welcome to the Metro

The Metro is a train network, mostly underground, enabling people to travel through the city quickly and conveniently.

The network is made of “lines”; each line connects a set of “stations”, two of which are its “end stations”. Trains on a line travel from one of the end stations to the other, stopping at each station along the way, and then back in the same manner.

Some stations belong to two or more lines; they are called “exchanges” and allow passengers to connect from one line to another.

To go to a certain destination using the Metro you'll first identify the stations closest to where you are and to where you want to go, then you'll find a Metro route between them; the route is made of a number of segments, each consisting of successive stations on a single line; successive segments connect through exchange stations.

This is probably more pompous than what you would tell a visitor who's not used the Metro before, but still far less precise and complete than what we expect from the “requirements document” of a software project in industry. It's good enough for our purposes of discovering a few classes.

First class ideas

As usual in requirements documents, many details are irrelevant for our immediate needs, for example that the network runs “mostly underground”. The word “network” itself is not that useful. But without much hesitation we can spot four concepts likely to yield classes:

- ***METRO_STATION***. The Metro is made of stations; people travel from a station to a station, going through other stations. This seems like an inevitable notion for our software.
- ***METRO_LINE***: the Metro consists of a set of lines, each connecting a number of stations that the line traverses in a set order.
- ***ROUTE***: a description of how to go from a given station to another.
- ***SEGMENT***: a set of contiguous stations on a line.

Close relations exist between these notions: a line is made of stations; a segment, also made of stations, is part of a line; a route is made of segments.

You indeed have at your disposal, in the TRAFFIC software, a set of classes covering these notions, and we may now take a look at some of their properties. Even though the classes are available as part of the material for this book, we'll work initially as if *we* had to design the corresponding classes, starting from the basic concepts of line, segment, station and route.

What characterizes a metro line

Let's start by understanding the interface (in the sense of program interface, or API) of a class representing lines. You can use the EiffelStudio environment to see the interface of any available class, also known as its **contract view** for reasons that will become clear as we go.

Let's first look at a simplified form of class *METRO_LINE*, called *SIMPLE_LINE*. Bring up the contract view of this class (the EiffelStudio appendix gives the instructions). The result looks like as follows:

→ "[*BRINGING UP A CONTRACT VIEW*](#)", [*A.4, page 575*](#).

***A class
interface***

The contract view shows the features — commands and queries — applicable to an instance of a class *SIMPLE_LINE*, representing a line of the metro. We'll study them in the next two sections.

When you have read the discussion of these features, go back to the preceding picture (or display again the corresponding contract view on your computer) and make sure you understand *all* that it shows.

To follow the discussion of queries and commands you need to remember that the class describes a set of possible objects, its instances (themselves representing individual lines of the metro). The features are defined in the class, but each defines an operation applicable to any such object. For example the class will have a query *sw_end* giving one of its two end stations (the one to the South or to the West of the other); this means that we may apply this query to any of its instance. If *Line8* denotes an instance of *SIMPLE_LINE*, then *Line8.sw_end* denotes its end station. We commonly say things like “A *SIMPLE_LINE*” to talk about a typical instance of the class, representing a typical line.

SIMPLE_LINE represents a slightly simplified version of the final class *METRO_LINE*; the discussion applies to both classes. We'll look at the queries first, then the commands.

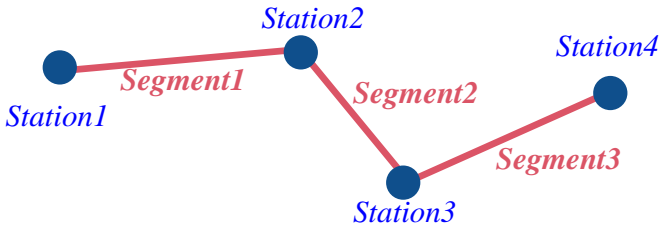
4.4 QUERIES

How long is this line?

One of the first things we may need to know about a line is the number of its stations. This is provided by a query *count*. The specification of that feature in the contract view appears as

```
count: INTEGER
-- Number of stations in this line
```

The second line is, as you know, a comment, more precisely a **header comment** that should come with every feature. It's always useful to give a plain language explanation of what a feature is about. Among other things this avoids misunderstandings. Here for example, we could have chosen to count the number of segments (elementary segments, those from a station to the next) rather than stations; the result would always be one fewer, as illustrated on this little line with four stations and three elementary segments:



*Four stations,
three segments*

The comment clarifies our convention: for class *SIMPLE_LINE* the *count* of a line is the number of its stations.

In the comment, note the expression “in this line”. The class describes the general notion of line, but when a feature like *count* is applied to a particular line, as in the feature call *Line8.count*, it will give us the station count of that line. So in the end the class always talks about a particular line, even though in the class we don’t know what it is. That’s what “this line” means: whatever line object to which we will apply the feature *count*.

The query declaration starts with: *count: INTEGER*. This simply introduces the name of the query, *count*, and the type of the result it returns, *INTEGER*. A query is there to provide information on an object (here an instance of *SIMPLE_LINE*), so the interface of the class must say what type of information that is.

INTEGER is such a type, denoting integer values, zero, positive or negative. The names of types, like classes, will always be written all in upper case. Other types encountered later in this chapter include:

- *STRING*, for values that are sequences of characters, such as “ABCDE”.
- *BOOLEAN*, for “truth values” that can only be either *True* or *False*.
- Classes themselves, such as *METRO_STATION* or *SEGMENT*.

More on types soon. For the moment *INTEGER* suffices.

Experimenting with queries

As you encounter features in this chapter, you can try them out.

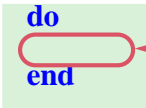
Programming time! **Length of a line**

The first programming exercise of this chapter, detailed below, lets you find the length of Line 8 of the Metro.

A system called *Metro* has been set up as part of the TRAFFIC software. Start EiffelStudio now on that system and bring up the text of its class *QUERIES* (see [instructions](#) in the EiffelStudio appendix). This is just a playground for trying out the concepts of this chapter; to achieve this you can, as you go along, fill in with various feature calls the part highlighted below. You'll be able to execute the resulting system and see the effects in each case.

→ "[BRINGING UP A CONTRACT VIEW](#)", A.4, page 575.

```

class QUERIES inherit
    TOUR
feature
    make is
        -- Try out queries and commands on lines.
        do
            
        end
end

```

The Metro Line 8 is defined, in the context set up for you by class *TOUR*, by a feature called *Line8*. Enter, into the “fill-in” part, the instruction

```

Console.show (Line8.count)

```

This calls the just described feature *count* on *Line8*, and then uses the command *show* on *Console* to display the result in the console window. Now you know how many stations Line 8 has.

As in the discussion of objects there's still a little “magic” involved since you are relying on *Console* and on *Line8* (denoting an instance of *SIMPLE_LINE*), both prebuilt for you in class *TOUR*. There will be a little more such magic in this chapter; we need it to let you concentrate on the new concepts you are learning. The goal, of course, is to remove the magic, and pretty soon you'll be able to define everything you need.

From the terminology of the chapter on objects you will remember that *Line8.count*, denoting the result of applying a query to an object, is an **expression**. Every expression has a type; here, because the query *count* has been declared to return an *INTEGER* result, the type of the expression is also *INTEGER*, as appropriate since it denotes a number of stations.

The stations of a line

Our next queries tell us exactly what stations are on a line. Remember the explanation in our little requirements statement:

... each line connects a set of “stations”, two of which are its “end stations”...

Although we need to complement this imperfect specification with our intuitive understanding of a transportation network, it clearly implies that a line contains a *sequence* of stations: first end station; a station; another station; and so on up to the other end station. A simple way to represent this is the following query of class *SIMPLE_LINE*:

```
i_th (i: INTEGER): METRO_STATION
-- The station of index i on this line
```

The name *i_th* comes from the common way of referring to an item by its position in a series: “the *i*-th element”, as in “the 25-th element”. We couldn’t call the query *i-th* because hyphens “-” are not permitted in identifiers, but underscores “_” are.

← “[Syntax: Identifiers](#)”, page 48.

The query *i_th*, like *show* for *Console*, takes an argument, representing the number, or “index” of the stop we want, starting at 1 for the first end station, then 2 for the first stop after it and so on. So if we again take Line 8 as an example, and refer to this map extract:

← “[FEATURES WITH ARGUMENTS](#)”, 2.4, page 34.



Start of line 8

To know all about Line 8 of the Metro:

www.chez.com/jefv/lignes/ligne8.htm

then we can use, in our program text, the expression

```
Line8.i_th (1)
```

representing the station called “Balard” on the above map); *Line8.i_th* (2) is “Lourmel” and so on. *Line8* has been predefined for us as part of the “magic”.

For consistency we take the (arbitrary) convention of starting the numbering of stations, for every line, at the South or West end.

Quiz time: The other end

`Line8.i_th (1)` is an expression of type `METRO_STATION` denoting the station at the Southwest end of Line 8. Without looking up the number of stations on that line or the names of individual stations (or the answer to this quiz, which appears a few paragraphs down), write another expression that denotes the object representing the station at the *other* end of the line. *Hint*: use another query already introduced.

Class `SIMPLE_LINE` has the following two queries denoting the ends of a line

```
sw_end: METRO_STATION
    -- End station on South or West side

ne_end: METRO_STATION
    -- End station on North or East side
```

← Query `sw_end` was mentioned on page 58.

Properties of start and end lines

To express more precisely our decision to start numbering at the Southwest end, we note that any line `l` will satisfy the following properties:

```
l.sw_end = l.i_th (1)
l.ne_end = l.i_th (l.count)
```

Don't even *think* of reading any further unless you understand these two lines perfectly. Each states a property of `l`, an equality, similar to equalities you have seen in mathematics, such as $\cos^2(x) + \sin^2(x) = 1$ for any number `x`:

- The first equality says that the query `sw_end` will always return the same result as the query `i_th` applied to the same metro line with the argument 1; in other words, it states our convention that station numbering on a line starts at the Southwest end.
- The second equality gives the corresponding equality at the other end. Since `l.count` denotes the number of stations on the line, the expression `l.i_th (l.count)` denotes the last station.

This also gives us the answer to the little quiz above: the expression denoting the end station of Line 8 is `Line8.i_th (Line8.count)`.

4.5 COMMANDS

So far we have accessed properties of existing lines, using queries. It's time now to look at the other category of features: commands, which enable us to change an object.

Building a line

What can we do to a line that will change it? The most obvious operation is to add a station to it, for example at the end.

If you are thinking: “*This is nonsense: a program cannot create a Metro station, and the Metro lines already exist anyway!*”, you should probably read again the [section](#) that explained that our objects are **software** artefacts, not the real thing. We will need the ability to change lines, if only to set up our object structure at the beginning of an execution once we get rid of the magic of class *TOUR* which at the moment creates the structure for us.

← “[Objects you can and can't kick](#)”, page 28.

To set up the object structure ourselves we might get the information from some external description of the Metro structure (in a *file* or *database*), then use it to create all the objects we need such as stations and lines.

Let's indeed rebuild line 8. From class *TOUR* we may assume the following: predefined features such as *Station_Balard*, *Station_Lourmel* etc. are available for every station; the name of the feature for station “xxx” is *Station_Xxx*. (Multiple words, as with the station “Félix Faure” of Line 8, are separated by underscores in the identifier, and for simplicity all letters in the identifiers are unaccented: *Station_Felix_Faure*.)

Of course *Line8* is itself predefined from *TOUR*, so the first thing we need to do is to empty it out of its stations. In the above interface of class *SIMPLE_LINE* the following command will do the job:

```
remove_all_segments
-- Remove all stations except the South-West end.
```

Our client program will use it under the form

```
Line8.remove_all_segments
```

Note that by convention our lines will always have at least one station, the Southwest end; when it's the only one, for example after a call to `remove_all_segments`, it will be the value of both `sw_end` and `ne_end`.

Now we are ready to add stations. The relevant command in this case is, in contract view:

```
extend (s: METRO_STATION)
  -- Add s at end of line.
```

This means that if `li` denotes a line you may add a station `st` at its end through

```
li.extend (st)
```

Indeed, you may now try the following new class:

```
class COMMANDS inherit
  TOUR
feature
  make is
    -- Recreate a partial version of Line 8
  do
    Line8.remove_all_segments
      -- No need to add Station_Balard, since
      -- remove_all_segments retains the SW end.
    Line8.extend (Station_Lourmel)
    Line8.extend (Station_Boucicaut)
    Line8.extend (Station_Felix_Faure)
      -- We stop adding stations, to display some results:
    Console.show (Line8.count)
    Console.show (Line8.ne_end.name)
  end
end
```

Quiz time: The last name shown

As you may guess from the last instruction, class `METRO_STATION` (the type of `ne_end`) has a query `name`, which gives the name of a station. What name should this last instruction display in the console window?

To check that your reasoning is correct, run this example now.

→ For instructions, see
[“RUNNING QUERIES
 ON A SYSTEM”, A.5.](#)
 page 575.

4.6 CONTRACTS

One of the reasons that the “line” class *SIMPLE_LINE* used so far is not the final *METRO_LINE* class is that it misses a fundamental property that we can’t ignore if we are to write serious software: that not all features are applicable to every possible argument and instance. Interfaces will need to be more precise about what is permitted.

Preconditions

The interface for the query *i_th* in class *SIMPLE_LINE*, as shown earlier

```
i_th (i: INTEGER): METRO_STATION
    -- The i-th station on this line
```

doesn’t mention that not every value for *i* makes sense: the value must be between 1 and the number of stations on the line, *count*. If Line 8 has 20 stations [CHECK NUMBER] then it would make no sense to use *Line8.i_th* (300), or *Line8.i_th* (0), or *Line8.i_th* (−1).

You may try such an out-of-bounds value on the computer if you wish, and see what happens.

A programmer who is trying to understand what the class is about — a potential “*client programmer*” — needs this kind of information. That’s precisely what interfaces are about: telling client programmers what classes can do for them.

We could of course add the information to the header comment, as in

```
i_th (i: INTEGER): METRO_STATION
    -- The i-th station on this line
    -- (Warning: use only with i between 1 and count, inclusive.)
```

*NOT RECOM-
MENDED STYLE*
see next..

which is better than nothing, but not good enough. Such usage properties are so common, and so critical for the proper use of classes and their features, that they must be treated as an integral part of the program, at the same level as the instructions and expressions. They will be called **contracts**. For *i_th* we have our first form of contract, the **precondition**. A precondition is a property that a feature imposes on all its clients; here, it’s that the argument must be within a certain range.

The interface of a feature will show the contract using the keyword **require**. So the contract view of class *METRO_LINE* actually describes *i_th* in this way:

```
i_th (i: INTEGER): METRO_STATION
    -- The i-th station on this line
require
    not_too_small: i >= 1
    not_too_big: i <= count
```

The precondition clause is made of two separate elements called **assertions**. Each expresses a property: $i \geq 1$ in the first assertion and $i \leq \text{count}$ in the second one. Note that because of the limitations of computer keyboards we can't use the mathematician's symbols \geq and \leq ; programming languages let us use instead 2-character symbols $>=$ and $<=$. Also, the names `not_too_small` and `not_too_big`, called **assertion tags**, serve to clarify the purpose of the assertions, but the real meaning is in the expressions that follow, $i >= 1$ and $i <= \text{count}$. We may omit the assertion tags and colons, as in

```
require
    i >= 1
    i <= count
```

without changing the meaning of the precondition, but it's clearer with the tags. When present, the tags appear in **roman** to stand out from the program elements in *italics*.

Expressions like $i >= 1$ and $i <= \text{count}$ denote *conditions* that, at any time during program execution, may be either true or false. Earlier examples involved equality, as $l.\text{sw_end} = l.i_th(1)$ for a line *l*. An expression that can take the values true and false — written in Eiffel as *True* and *False*, with a capital first letter since they are predefined values — is known as **boolean**:

← Page 62.

Definition: Boolean value

A boolean value is one of: *True* and *False*.

The corresponding type is called *BOOLEAN*; it's one of the types we have at our disposal, along with *INTEGER*, *STRING* and class names. Most other types have many values — we'll see for example that typically the representation for an integer value on a computer supports some four billion possibilities — but *BOOLEAN* provides only two. The purpose is clearly to represent the notion of condition, similar to non-programming uses of this concept: in “You can go skiing only if there is enough snow”, the property “there is enough snow” is a condition — a boolean expression. As usual, our boolean expressions in software must be more precisely defined, like in mathematics: $i \geq 1$ is unambiguously true or false once we know the value of the integer i , whereas how much snow is “enough snow” is subject to human judgment.

← The notion of type was introduced in “How long is this line?”, page 58.

Boolean values lie at the heart of *logic*, the art of reasoning; the next chapter is devoted to this topic. Preconditions and the other forms of contract will use boolean expressions to state conditions that clients and suppliers must satisfy. Here the precondition of *i_th*, as it appears in the interface

require

not_too_small: $i \geq 1$
not_too_big: $i \leq \text{count}$

is essential information for the client.

A client that does *not* satisfy that property, for example if it has a call

Line8.i_th(1000)

is faulty software, or *buggy* according to habitual terminology, where a “bug” is simply an error.

We may express this observation as a general principle:

Touch of Methodology: Precondition Principle

A client calling a feature must make sure that the precondition holds before the call.

Whenever you consider using a feature, you will see its specification in the contract view of the corresponding class, including its precondition if any, as in the example of *i_th* above. It is then your responsibility, as the client programmer, to make sure that any call to the feature satisfies the precondition.

Some features are always applicable; they do not have a **require** clause. By convention this means the same as if they had one of the form

```
require  
  always_OK: True
```

defining a precondition that is always satisfied.

Contracts for debugging

One way that preconditions and other contracts will help you during software development is that the tools will check them when you execute your program. So if one of the contracts does not hold, revealing a bug, you will get a precise message telling you what happened; the message lists the tag (such as `not_too_small`) of the violated assertion, so that you know what exactly went wrong.

When a program is ready for distribution, you should have corrected all the bugs, and can change the options to stop checking contracts at run time.

Contracts for interface documentation

The better approach to software correctness is, of course, to avoid bugs in the first place (rather than make mistakes and then correct them); systematic use of contracts helps. In particular, the documentation of a software mechanism, as given by its interface, should always list the complete *precondition* that defines under what circumstances it is legitimate to use the mechanism.

This style — illustrated by the interface for *i_th* as shown above — will be the standard form of interface description for the rest of this book.

Postconditions

In describing the interface that a feature presents to its potential clients, preconditions address only one side: what a feature expects from the clients before a call. For the clients, a precondition is an *obligation*. As in any good relationship, the clients will want to know what *benefits* they will get after a call. The feature's interface can express this through a **postcondition**.

Unlike with preconditions, we won't always be able in postconditions to express all relevant properties, but often we can say something interesting anyway. Here for example is the interface for *remove_all_segments* in class *ROUTE*:

```

remove_all_segments
  -- Remove all stations except the South-West end.
ensure
  only_one_left: count = 1
  both_ends_same: sw_end = ne_end

```

Here there is no precondition since *remove_all_segments* is always applicable to a route. The keyword **ensure** introduces a postcondition. Here the feature guarantees two things to its client when it has done its job:

- The number of stations, *count*, is equal to 1.
- The two end stations, *sw_end* and *ne_end*, are now the same station.

Similarly, here is the interface (precondition omitted) for *extend*, the command that adds a station at the end of a line:

```

extend (s: METRO_STATION)
  -- Add s at end of line.
ensure
  new_station_added: i_th (count) = s
  added_at_NE: ne_end = s
  one_more: count = old count + 1

```

The first postcondition clause uses the query *i_th*: it states that after a call to *extend*, if we ask what is the station at position *count*, that is to say the last station, the answer will be *s*, the station that we have asked the command *extend* to add. This states precisely the intent of *extend*; if the command does its job properly — that is to say, if its program text doesn't have any bugs — this property will always hold as a result of an execution of the command.

The second clause expresses that *ne_end* will also be equal to *s*. The invariant, to be seen in the next section, will tell us that *ne_end* must be equal to *i_th (count)*, so this clause is in fact redundant, but it doesn't hurt.

The third clause tells us that the routine increases *count* by one. It uses a keyword that we haven't encountered yet, **old**. A postcondition clause states a property that will hold when a routine call terminates; it often needs to relate the value an expression then has to the value it had on *entry* to the procedure. Hence the usefulness of an “Old expression”, of the form

```
old some_expression
```

which means: “The value of *some_expression*, captured at the beginning of the routine's execution”. Here the postcondition clause

```
count = old count + 1
```

states that the routine must increase the number of stations, *count*, by one.

Old expressions, and the **old** keyword, may only appear in postconditions.

When you write a feature in a class, you may assume that the precondition holds at the beginning, but it is your responsibility to ensure that the postcondition holds at the end of the feature's execution:

Touch of Methodology: Postcondition Principle

A feature must make sure that, if its precondition held at the beginning of its execution, its postcondition will hold at the end.

Class invariants

Preconditions and postconditions are logical properties of your software, each associated with a particular *feature*, such as *i_th*, *remove_all_segments* and *extend* in the examples we've seen.

We also use logical properties to characterize an entire *class*, above the level of its individual features. Such a property, known as a **class invariant**, expresses relationships between the different queries of a class. As an example we saw earlier that if *l* is a line then $l.sw_end = l.i_th(1)$ and $l.ne_end = l.i_th(l.count)$. These are of course properties not of any particular *l* but of the class as a whole; indeed they figure in the class invariant, which appears at the end of the text of class *METRO_LINE*:

← "[Properties of start and end lines](#)", page 62.

invariant

```
southwest_is_first: sw_end = i_th(1)
northeast_is_last: ne_end = i_th(count)
```

This is typical of the role of class invariants: expressing consistency requirements between the queries of a class. We see here that in class *METRO_LINE* there is a certain redundancy between these queries: *sw_end* and *ne_end* provide information that one can also get through *i_th*, applied to arguments *1* and *count*.

Another example is a class *CAR_TRIP* providing queries such as *initial_odometer_reading*, *trip_time*, *average_speed* and *final_odometer_reading*, with roles implied by their names (“odometer reading” is the total number of kilometers or miles traveled). There is again a certain redundancy which you may capture through a class invariant (where * denotes multiplication):

invariant

consistent: $final_odometer_reading = initial_odometer_reading + trip_time * average_speed$

There is nothing wrong in principle with including such redundant queries when you design a class: they may all be relevant to the clients, even if they are derived from some of the same internal information about the corresponding objects. But without the invariant, the redundancy might cause confusion or errors. The invariant expresses clearly and precisely how the different queries may be related.

We saw earlier that a precondition must hold at the beginning of a feature call, and a postcondition at the end. An invariant — which applies to all the features of a class, not just a specific one — must hold at both points:

Touch of Methodology: Invariant Principle

A class invariant must hold as soon as an object is created, then before and after the execution of any of the features of the class available to clients.

Contracts: a definition

We have seen various kinds of contract — preconditions, postconditions, class invariants — from which a general definition now emerges:

Definition: Contract

A contract is a specification of properties of a software element that affect its use by potential clients.

We will use contracts throughout the software to make it clear what each element — class or feature — is about. As noted, they serve for documenting software, especially libraries of components meant (like the TRAFFIC software) for reuse by many different applications; they help in debugging; and they help us avoid bugs in the first place by writing correct software.

4.7 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A software element presents to the rest of the world one or more interface.
- Classes exist only in the software text; objects exist only during the execution of the software.
- A class describes a category of possible objects.
- Every query returns a result of a type specified in the query’s declaration.
- We may specify the interface of a class through a “contract view” which lists all the features of the class — commands and queries — and, for each of them, the properties relevant to *clients* (other classes that use it).
- A feature may have a precondition, specifying initial properties under which it is legitimate to call the feature, and a postcondition, specifying final properties that it guarantees when it terminates.
- A class may have a class invariant, specifying consistency conditions that connect the values of its queries.
- Preconditions, postconditions and class invariants are examples of contracts.
- Among other applications, contracts help for software design, for documentation, and for debugging.

4-E.8 New vocabulary

API	Assertion	Assertion tag	
Boolean invariant	Bug	Buggy	Class
Contract	Client	Client Programmer	
GUI	Design	Generating class	
Postcondition	Implementation	Instance	Interface
Supplier	Precondition	Program interface	
	Type	User interface	

4-E EXERCISES

4-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

4-E.2 Violating a contract

- 1 • Write a simple program that uses the query *i_{th}* of class *METRO_LINE*. Run it, using a known *METRO_LINE* object, for example *Line8*.
- 2 • Change the argument to *i_{th}* so that it’s out of bounds (less than one, or larger than the number of stations). Run the program again. What happens? Explain the messages that you get.

5

Just Enough Logic

Programming is, for a large part, reasoning. We use computers to perform certain combinations of basic tasks, executed at rates beyond direct human comprehension; to get the results that we need, we must be able to understand the program's possible run-time behaviors, which are nothing but consequences and ramifications of the effects prescribed by our programs, if often very indirect consequences and many ramifications. All can, in principle, be deduced from the program text through mere reasoning. It would help us considerably if there were a science of reasoning.

We're in luck, because there *is* such a science: Logic. Logic is the machinery behind the human aptitude to reason. The laws of logic enable us, when told that Socrates is human, and that all humans are mortal, to deduce without blinking that Socrates, then, must be mortal. When someone announces that whenever the temperature in the city rises above 30 degrees a pollution alert will result, so because the temperature today is only 28 degrees there won't be a pollution alert, you'll say that his logic is flawed.

Logic is the basis of mathematics; mathematicians will believe a 5-line or 60-page proof only because they accept that each step proceeds according to the rules of logic.

Logic is also at the basis of software development. Already in the last chapter we encountered *conditions* in the contracts associated with our classes and features, for example the precondition “*i* must be between 1 and *count*”. We will also use conditions in expressing the actions of a program, for example “If *i* is positive, then execute this instruction”.

We've seen in the study of contracts how such conditions appear in our programs in the form of “boolean expressions”. A boolean expression may be complex, involving operators such as “**not**”, “**and**”, “**or**”, “**implies**”; this mirrors modes of reasoning familiar in ordinary language: “If it's already 20 minutes past the time for our date *and* she did *not* call *or* send an SMS, it *implies* she won't show up at all”. We all intuitively understand what this means, and so far this informal understanding has been good enough for our software conditions too.

No longer. Software development requires precise reasoning, and precise reasoning requires the laws of logic. So before we can plunge back into the delights of objects and classes we must familiarize ourselves with these laws.

Logic — *mathematical* logic as it is more precisely called — is a discipline of its own, and even just “Logic for Computer Science” is the topic of many textbooks and courses; I hope that you will take such a course or have already taken it. This chapter introduces some essential elements of logic needed to understand programming. More precisely, even though logic in its full glory is the science of reasoning, we need it, just now, for a more limited goal: understanding the part of reasoning having to do with *conditions*. Logic will give us a solid basis for expressing and understanding conditions as they appear in contracts and elsewhere in programs.

5.1 BOOLEAN OPERATIONS

A condition in logic as well as in programming languages is expressed as a boolean expression, built out of boolean variables and operators, and representing possible boolean values.

Boolean values, variables, operators and expressions

There are two **boolean values**, also called “truth values”; we write them as **True** and **False** for compatibility with our programming language, although logicians often use just **T** and **F**. In electrical engineering, which relies on logic for circuit design, they are often called 1 and 0.

→ **True** and **False** are “reserved words” of the programming language; see page [219](#).

A **boolean variable** is an identifier denoting a boolean value. Typically we use a boolean variable to express a property that might be either true or false: to talk about the weather we might have the boolean variable *rain_today* to stand for the property that we think rain will fall today.

Starting from boolean variables we may use **boolean operators** to make up a **boolean expression**. For example, if *rain_today* and *cuckoo_sang_last_night* are boolean variables, then the following will be boolean expressions according to the rules studied next:

- *rain_today*
-- A boolean variable, without operators: already a boolean expression (the simplest form).
- **not** *rain_today*
-- Using the boolean operator **not**.
- (**not** *cuckoo_sang_last_night*) **implies** *rain_today*
-- Using the operators **not** and **implies**, and parentheses -- to delimit a subexpression.

Each boolean operator — such as **not**, **or**, **and**, **=**, **implies** as defined below — comes with rules defining the value of the resulting expression from the values of the variables making it up.

For compatibility with the way we write programs, we express the boolean operators through the corresponding programming language keywords. In mathematical textbooks you will see them expressed as symbols, most of which you couldn't directly type on your keyboard. Here is the correspondence:

Eiffel keyword	Common mathematical symbol
not	\sim or \neg
or	\vee
and	\wedge
=	\Leftrightarrow or $=$
implies	\Rightarrow

Negation

The first operator is **not**. To use it to form a boolean expression, apply it to another existing expression, for example a single boolean variable, as in **not** *your_variable*, or a more complicated one, such as **not** (**not** *a*), or **not** (*a* **or** *b*), where *a* and *b* are boolean variables.

For an arbitrary boolean variable a , the value of **not** a is **False** if the value of a is **True**, and **True** if the value of a is **False**. We may also express this, the defining property of **not**, through the following table:

a	not a
True	False
False	True

This is called a **truth table** and is the standard way to specify the meaning of a boolean operator: in the first columns, list all the possibilities for the values of the variables involved in an expression that uses the operator; in the last column, list the corresponding value of the expression in each case.

The operator **not** represents **negation**: replacing every boolean value by its opposite, where **True** is the opposite of **False** and conversely.

From the truth table we note interesting properties of this operator:

Theorems: Negation properties

For any boolean expression e and any values of its variables:

- 1 • Exactly one of e and **not** e has value **True**.
- 2 • Exactly one of e and **not** e has value **False**.
- 3 • One of e and **not** e has value **True**. (**Principle of the Excluded Middle**.)
- 4 • Not both of e and **not** e have value **True**. (**Principle of Non-Contradiction**.)

Proof: by definition of a boolean expression, e can only have value **True** or **False**. The truth table shows that if e has value **True**, then **not** e has value **False**; all four properties are consequences of this (and particularly the last two directly from the first).

Disjunction

The operator **or** uses two operands (rather than one for **not**) If a and b are boolean expressions, the boolean expression a **or** b has value **True** if and only if at least one of a and b has that value. Equivalently, it has value **False** if and only if the operands both have that value. The truth table expresses this:

<i>a</i>	<i>b</i>	<i>a or b</i>
True	True	True
True	False	True
False	True	True
False	False	False

The first two columns list all four possible combinations of values for *a* and *b*.

The word “or” is taken here from ordinary language in its **non-exclusive** sense, as in “*Whoever made this regulation must have been stupid or asleep*”, which doesn’t rule out that he might have been both.

In ordinary language “or” is also frequently used in an *exclusive* sense, meaning that for the result to hold one of the conditions must hold, but not both, as in “*Shall we order red or white?*”. This corresponds to a different operator, “exclusive or” — **xor** in Eiffel — whose properties you are invited to study by yourself.

→ This is exercise
5-E.10, page 106.

The **or** operator, non-exclusive, is called **disjunction**. That’s not such a good name, because it may misleadingly suggest an exclusive operator; but it has the benefit of symmetry with “conjunction”, the name for our next operator, **and**.

A disjunction has value **False** in only one case out of the four possible value combinations: the last row in the table. This provides an alternative, often useful form of the definition:

Theorem: Disjunction Principle

An **or** disjunction has value **True** except if both operands have value **False**.

The truth table shows that the operator **or** is also *Commutative*: for any *a* and *b*, the value of *a or b* is the same as that of *b or a*. This is also a consequence of the Disjunction Principle.

Conjunction

Like **or**, the operator **and** takes two operands. If *a* and *b* are boolean expressions, then the boolean expression *a and b* has value **True** if and only if both *a* and *b* have that value. Equivalently, it has value **False** if and only if at least one of the operands both have that value. In truth table form:

<i>a</i>	<i>b</i>	<i>a and b</i>
True	True	True
True	False	False
False	True	False
False	False	False

The application of **and** to two values is known as their **conjunction**, as in the conjunction of two events: “*Only the conjunction of a full moon and Saturn’s low orbit can bring true romance to a Sagittarius*” (perhaps not, however, the kind of example that directly influences mathematical logicians).

Studying **and** and **or** reveals a close correspondence, or **duality**, between the two operators: many interesting properties of either operator yield a property of the other if we swap **True** and **False**. For example the Disjunction Principle has a dual that applies to conjunction: ← Page 77.

Theorem: Conjunction Principle

An **and** conjunction has value **False** except if both operands have value **True**.

Like **or**, the operator **and** is commutative: for any *a* and *b*, *a and b* has the same value as *b and a*. This property can be seen on the truth table; it’s also a consequence of the Conjunction Principle.

Complex expressions

You may use boolean operators — the three already introduced, **not**, **or** and **and**, and the other two described next — to build a more complex boolean expression, and deduce the truth table of the expression from the truth tables defining the operators. Here for example is the truth table for the boolean expression *a and (not b)*:

<i>a</i>	<i>b</i>	not b	<i>a and (not b)</i>
True	True	False	False
True	False	True	True
False	True	False	False
False	False	True	False

To derive this truth table, it suffices to replace, in the truth table for **and**, each value of *b* by the value of **not b** as obtained from the truth table for **not**; a third column has been added to show **not b**.

Truth assignment

A boolean variable represents a value that may be either **True** or **False**. The value of a boolean expression depends on the value of its variables. For example by building the truth table for a **and** (b **and** (**not** c)) you would see that this expression has:

- Value **True** if a has value **True**, b also, and c has value **False**.
- Value **False** in all other cases.

The following notion helps express such properties:

Definition: Truth assignment

A truth assignment for a set of variables is a particular choice of values, **True** or **False** in each case, for each one of the variables.

So we can say that a **and** (b **and** (**not** c)) has value **True** for exactly one truth assignment of its variables (the one that chooses **True** for a , **True** for b , and **False** for c) and **False** for all other truth assignments.

Each row of the truth table for an expression corresponds, one to one, to a truth assignment of its variables.

It is easy to see that for an expression involving n variables there are 2^n possible truth assignments and hence 2^n rows in the truth table. For example the table for **not**, with one operand, had $2^1 = 2$ rows; for **or**, with two operand, there were $2^2 = 4$ rows. The number of columns is $n + 1$:

- The first n columns of each row list the values of each of the variables for the corresponding truth assignment.
- The last column gives the expression's value for that truth assignment.

If an expression has value **True** for a certain truth assignment (as reflected in the last column for the corresponding row), we say that the truth assignment **satisfies** the expression. For example the truth assignment cited — **True** for a , **True** for b , **False** for c — satisfies a **and** (b **and** (**not** c)), all others don't.

Tautologies

We are often interested in expressions that have value **True** for every truth assignment of their variables. Consider

a or (not a)

This states that for a variable a either (or both):

- a has value **True**
- **not** a has value **True**.

This is only an informal interpretation; to study the value of this expression we may build its truth table, deduced from those for **or** and for **not**:

a	not a	a or (not a)
True	False	True
False	True	True

The second column is not strictly part of the truth table but gives the value of **not** a , coming from the table for **not**. Combining this with the truth table for **or** (which tells us that both **True or False** and **False or True** have value **True**) yields the third column.

From that column we see that any truth assignment — meaning here, since there's only one variable, any value of a , **True** or **False** — satisfies the expression. Such expressions have a name:

Definition: Tautology

A tautology is a boolean expression that has value **True** for every possible truth assignment of its variables.

The property that a or (not a) is a tautology was expressed earlier as the **Principle of the Excluded Middle**.

← Page 76.

Other simple tautologies, which you should now prove by writing their truth tables, are:

- **not** (a and (not a)), expressing the Principle of Non-Contradiction.
- (a and b) or ((not a) or (not b))

← Also page 76.

Sometimes it's also interesting to exhibit a property that is *never* true:

Definition: Contradiction

A contradiction is a boolean expression that has value **False** for every possible truth assignment of its variables.

For example (check the truth table again), **a and (not a)** is a contradiction; this restates, more simply, the Principle of Non-Contradiction.

From these definitions and the truth table for **not** it follows that a is a tautology if and only if **not a** is a contradiction, and conversely.

An expression that has value **True** for at least one truth assignment of its variables is said to be **satisfiable**. Obviously:

- Any tautology is satisfiable.
- No contradiction is satisfiable.

There exist, however, satisfiable expressions that are neither tautologies nor contradictions: they have value **True** for at least one truth assignment, and value **False** for at least one other truth assignment. This is the case, for example, with **a and b** and with **a or b** .

“ a is not a tautology” isn't the same as “**not a** is a tautology”. The second property states that no truth assignment satisfies a or, as just seen, that a is a contradiction. The first property states that at least one truth assignment doesn't satisfy a ; but then some other truth assignments might still satisfy a , in which case a is satisfiable but neither a tautology nor a contradiction.

Equivalence

To prove or disprove tautologies, contradictions and satisfiability, we're soon going to get fed up with writing truth tables. With 2^n rows for n variables, this is tedious; to find that **a and (b and (not c))** is satisfiable but neither a tautology nor a contradiction we would have to consider 8 cases. We need a better way. For example, you may have resented being asked to use a truth table to show that **a and (not a)** is a contradiction if previously you had proved that **not (a and (not a))** is a tautology. It's time for more general rules.

The equivalence operator helps in defining such rules. It uses the equal symbol, $=$, and has the following table (the truth table to end all truth tables!) stating that $a = b$ has value **True** if and only if a and b either have both the value **True** or both the value **False**:

a	b	$a = b$
True	True	True
True	False	False
False	True	False
False	False	True

This operator is commutative ($a = b$ always has the same value as $b = a$). It is also *reflexive*, that is to say, $a = a$ is a tautology for any a . → See exercise [5-E.2](#), page 104.

Although logicians usually use the symbol \Leftrightarrow for equivalence, the equality symbol $=$ is appropriate because $a = b$ really expresses equality in the usual sense, denoting an expression that has value **True** if and only if a and b have the same value. The following property extends this observation:

Theorem: Substitution

For any boolean expressions u , v and e , if $u = v$ is a tautology and e' is the expression obtained from e by replacing every occurrence of u by v , then $e = e'$ is a tautology.

Proof sketch: if u doesn't occur in e , then e' is the same expression as e , and we have seen (reflexivity of $=$) that $e = e$ is a tautology. If u does occur in e , we note that the value of a boolean expression under any particular truth assignment is entirely determined by the value of its sub-expressions under that assignment. Here e' differs from e only by having occurrences of the sub-expression u replaced by v . Under any particular truth assignment, since $u = v$ is a tautology, these sub-expressions will have the same value in e and e' ; because the rest of the expression is the same, the value of the entire expression will be the same, implying that the truth assignment satisfies $e = e'$. Since this is the case for any truth assignment, $e = e'$ is a tautology.

This rule is the key to proofs of non-trivial boolean properties. We do proofs by truth tables for only the basic expressions; then we use equivalences to replace expressions by simpler ones. For example, to prove that

$$(a \text{ and } (\text{not } (\text{not } b))) = (a \text{ and } b) \quad \text{/GOAL/}$$

is a tautology, you don't need to write its truth table; you prove first that for any expression x the following general properties are both tautologies:

$\text{not } (\text{not } x) = x$	/T1/
$x = x$	/T2/

/T2/ is the reflexivity of $=$, proved from the truth table; /T1/ is easily proved in the same way. We may use /T1/, applied to the expression b , to replace $\text{not } (\text{not } b)$ by just b on the left-hand side of the property /GOAL/; then applying /T2/ to a and b yields the desired property.

To express that two boolean values are *not* equal, we will use the symbol \neq (the best approximation, with two characters available on all keyboards, of the mathematical symbol \neq). Its definition is that $a \neq b$ has the same value as $\text{not } (a = b)$.

De Morgan's laws

Two tautologies are of particular interest in using **and**, **or** and **not**:

Theorems: De Morgan's Laws

The following two properties are tautologies:

- $(\text{not } (a \text{ or } b)) = ((\text{not } a) \text{ and } (\text{not } b))$
- $(\text{not } (a \text{ and } b)) = ((\text{not } a) \text{ or } (\text{not } b))$

Proof: either write the truth tables, or better combine the Non-Contradiction, Excluded Middle, Disjunction and Conjunction principles.

These properties make the **and-or** duality even more remarkable, by expressing that if you negate either of the two operators you get the other by negating the operands.

Informally interpreting — for example — the first one: “if we say that it's not true that a or b holds, it's exactly the same as if we were saying that neither a nor b holds”. Of course we're already at a stage where formal notations such as those of logic, with their precision and concision, become vastly superior to such natural-language statements.

Another aspect of the close association between the **or** and **and** operators is that each is **distributive** with respect to the other, meaning that the following two properties are tautologies:

Theorems: Distributivity of boolean operators

The following two properties are tautologies:

- $(a \text{ and } (b \text{ or } c)) = ((a \text{ and } b) \text{ or } (a \text{ and } c))$
- $(a \text{ or } (b \text{ and } c)) = ((a \text{ or } b) \text{ and } (a \text{ or } c))$

(Compare to the distributivity of multiplication with respect to addition in arithmetic: if $+$ is addition and $*$ is multiplication, then $m * (p + q)$ is the same as $(m * p) + (m * q)$ for any numbers m, p, q .) Distributivity is easy to prove, for example from truth tables. It helps simplify complex boolean expressions.

Simplifying the notation

To avoid the accumulation of parentheses, it is customary to accept some *precedence rules* that give a standard understanding for boolean expressions, removing ambiguity even if some parentheses are missing. This is the same idea that enables us to understand $m + p * q$, in arithmetic and in programming languages, as meaning $m + (p * q)$ rather than the other possible grouping. We say that the operator $*$ **binds tighter**, or has **higher precedence**, than the operator $+$: it “attracts” the neighboring operands before $+$ gets its chance.

For boolean operators we may use the same precedence as used by the syntax of Eiffel; the order from highest precedence to lowest is:

- **not** binds tightest.
- Then comes equivalence: $=$.
- Then comes **and**.
- Then **or**.
- Then **implies** (studied below).

Under these rules the expression $a = b \text{ or } c \text{ and not } d = e$, with no parentheses, is legal and means

$$(a = b) \text{ or } (c \text{ and } ((\text{not } d) = e))$$

It is desirable, however, to retain some parentheses to protect readers of your programs from misunderstandings which might lead to errors.

In the recommended style you should *not* drop the parentheses that separate **or** and **and** expressions since the precedence rule making **and** bind tighter than **or** is arbitrary. It is also better to keep the parentheses around a **not** subexpression used as operand of an equivalence, to avoid confusing $(\text{not } a) = b$ with $\text{not } (a = b)$. You may, however, drop the parentheses around a subexpression of the form $x = y$ where x and y are single variables. So for the last example you would just write

$$a = b \text{ or } (c \text{ and } (\text{not } d) = e)$$

The reason

Another property that simplifies the notation is the **associativity** of certain operators. In arithmetic we commonly write $m + p + q$ even though it could mean $m + (p + q)$ or $(m + p) + q$, because the choice doesn't matter: these two expressions have equal values, reflecting that addition is an *associative* operation. Multiplication is also associative: $m * (p * q)$ always has the same value as $(m * p) * q$. In boolean logic both operators **and** and **or** are associative, as expressed by the following tautologies:

$$\begin{aligned} (a \text{ and } (b \text{ and } c)) &= ((a \text{ and } b) \text{ and } c) \\ (a \text{ or } (b \text{ or } c)) &= ((a \text{ or } b) \text{ or } c) \end{aligned}$$

For the proofs: you may write truth tables but it's easier to use previous rules. In the first example, the left side is true, from the Conjunction Principle, if and only if both a and $b \text{ and } c$ are true, that is to say — applying that Principle again — if and only if all three of a , b and c are true; but from the same reasoning this is also the case with the right-hand side, so the two sides are equivalent (satisfied under exactly the same truth assignments). For the second line the reasoning is the same, using the Disjunction Principle.

This enables us to write expressions of the form $a \text{ and } b \text{ and } c$, or $a \text{ or } b \text{ or } c$, without risk of confusion. To summarize:

Touch of Style: **Parentheses for boolean expressions**

In writing subexpressions of a boolean expression, drop the parentheses:

- Around " $a = b$ " if a and b are single variables.
- Around successive terms if they each involve a single boolean variable and are separated by the same associative operators.

For clarity and to help avoid errors, retain other parentheses, without taking advantage of precedence rules.

5.2 IMPLICATION

One more basic operator — along with **not**, **or**, and **and** and equivalence — belongs to the basic repertoire: implication. Although it's similar to the others, and in fact close to **or**, it requires some attention because its precise properties initially seem, for some people, to contradict intuitive views of the concept of implication in ordinary language.

Definition

The simplest way to define the **implies** operator is in terms of **or** and **not**:

Theorem: Implication

The value of a **implies** b , for any boolean values a and b , is the value of
(**not** a) **or** b

→ This definition will be slightly generalized on page [96](#).

This gives the truth table (which could serve as an alternative definition):

a	b	a implies b
True	True	True
True	False	False
False	True	True
False	False	True

It's the same as the [table](#) for **or**, with **True** and **False** values for b switched. ← Page [77](#).
The result of a **implies** b is true for all truth assignments except in one case, the highlighted entry: when a is true and b false.

In a **implies** b the first operand a is called the **antecedent** of the implication, and the second operand b is called its **consequent**.

The principles we saw for conjunction and especially disjunction have a direct counterpart with implication:

Theorem: Implication Principle

An implication has value **True** except if its antecedent has value **True** and its consequent has value **False**.

In particular, it always has value **True** if the antecedent has value **False**.

Relating to inference

The name “**implies**” suggests that we can use the implication operator to infer properties from others. This is indeed permitted by the following theorem:

Theorem: Implication and deduction

- If a truth assignment satisfies both a and a **implies** b , it satisfies b .
- If both a and a **implies** b are tautologies, b is a tautology.

Proof: To prove the first clause, consider a truth assignment TA that satisfies a . If TA also satisfies a **implies** b , then it must satisfy b , since otherwise under row 2 of the truth table for **implies** the value of a **implies** b would be **False**. To prove the second clause, note that if a and a **implies** b are tautologies this reasoning is valid for any truth assignment TA .

← The highlighted entry on page [86](#).

This property legitimates the usual practice, when we want to prove a property b , to identify a possibly “stronger” property a , and prove separately that

- a holds.
- a **implies** b holds.

Then we may deduce that b holds.

The term “stronger” used here is useful in the practice of reasoning with contracts of programs, and deserves a precise definition:

Definitions: Stronger, weaker

For two non-equivalent expressions a and b , we say that:

- “ a is **stronger** than b ” if and only if a **implies** b is a tautology.
- “ a is **weaker** than b ” if and only if b is stronger than a .

The definitions assume a and b to be non-equivalent because it could be confusing to say that a is stronger than b if they might be the same. In such cases we’ll use “stronger than or equal to” and “weaker than or equal to” (as with relations between numbers: “greater than”, “greater than or equal to”).

Getting a practical feeling for implication

How does the definition of **implies** relate to the usual notion of implication, expressed in ordinary language by such locutions as “If ... then ...”?

In such everyday use, implication often indicates causality: “*If* we get any more sun, *then* this will be a vintage year for Bourgogne” suggests that one event causes another. The **implies** of logic does not connote causality, it simply states that whenever a certain property is true another one must be too. The example just given can also be interpreted this way if we ignore any hint of causality.

Another typical example is (at the Los Angeles airport, trying to check in for Santa Barbara): “*If* your ticket says Flight 3035, *then* you’re not flying tonight”, perhaps because the plane is grounded for mechanical problems and all other flights are full. There is no causality here: what’s printed on the ticket didn’t cause the plane to malfunction. It’s simply that for anyone for whom the property “Reserved flight is 2096” holds, the property “can fly today” doesn’t hold. Logic’s **implies** operator covers this.

What — surprisingly — surprises many people is the property stated at the end of the Implication Principle and resulting from the last two rows of the truth table: that whenever *a* is false the implication *a* **implies** *b* is true, regardless of the value of *b*. In fact this *does* correspond to the usual idea of implication:

- 1 • “*If I am the Pope, two plus two equal five*”
- 2 • “*If two plus two equal five, then I am the Pope*”
- 3 • “*If two plus two equal five, then I am not the Pope*”
- 4 • “*If I am the Pope, two plus two equal four*”
- 5 • “*If I am the Pope, it will rain today*”
- 6 • “*If it rains today, I will not be elected Pope before the end of the year*”

Given that I am not the Pope and don’t expect to run for the job this year, all these implications are true — regardless, for the last two, of today’s weather.

All that “If *a*, then *b*” tells us is that whenever the antecedent *a* holds, the consequent *b* must hold too. So the only possibility for this implication to be false is (second row, with highlighted entry, in the truth table) for *a* to be true and *b* false. Cases in which the antecedent does not hold, and cases in which the consequent holds, tell us nothing about the truth of the implication as a whole.

[Inclusion of pictures subject to pending request for permission.]

From:
"Explorers on
the Moon" by
Hergé, Ameri-
can edition ©
1976 Little,
Brown & Co.

See exercise
[5-E.7, page](#)
[105.](#)

Beginners sometimes have trouble with accepting that “ a implies b ” can be true if a is false; most of the trouble, I guess, comes from the case in which a is false and b is true — such as [1](#), [2](#) and possibly [5](#) and [6](#) above — although there is nothing wrong with it. In fact, the trouble may come from a common distortion of reasoning which leads some people, equipped with the knowledge that a implies b , to infer happily that if a doesn’t hold then b must not hold either! Typical examples:

- F1 • “All professional politicians are corrupt. I am not a professional politician, so I am not corrupt and you must vote for me”. If the premise is true it tells us something about professional politicians, but nothing at all about anyone else!
- F2 • “Whenever I take my umbrella it doesn’t rain, so I’ll leave my umbrella at home as we badly need some rain right now.” Joke of course, but suggesting the same flawed reasoning.
- F3 • “All recent buildings in this area have bad thermal isolation. This is an older building, so it must be more comfortable in hot summers”.

In each of the cases there’s a property a that implies another b , and it’s erroneously deduced that the negation of a implies the negation of b . But we can’t deduce any such thing. All we know is that if a holds then b will hold; if a doesn’t hold, knowledge of the implication tells us nothing interesting. Couched in the language of logic, the flaw is to believe that

$$(a \text{ implies } b) = ((\text{not } a) \text{ implies } (\text{not } b))$$

Warning: not a tautology (see exercise)

is a tautology. Or perhaps it’s just imagining the slightly less powerful $(a \text{ implies } b)$ implies $((\text{not } a) \text{ implies } (\text{not } b))$. Neither is a tautology, as they both have value **False** when a has value **False** and b has value **True**.

Reversing an implication

Although the last two properties are not tautologies, there is an interesting tautology of the same general style:

$$(a \text{ implies } b) = ((\text{not } b) \text{ implies } (\text{not } a)) \quad \text{/REVERSE/}$$

Proof: we just expand the definition of **implies**. For the left side, it gives $(\text{not } a) \text{ or } b$; for the right side, $(\text{not } (\text{not } b)) \text{ or } (\text{not } a)$. From a [previous](#) tautology, we know that $(\text{not } (\text{not } b))$ is b ; from the commutativity of **or**, the right side has the same value as the left side for any truth assignment.

← /TI/, page [83](#).

Alternatively, we may note in the truth table for **implies** that swapping a and b then negating both yields back the original table.

This property, */REVERSE/*, states that if b holds whenever a does, then from the knowledge that b doesn't hold we may infer that a doesn't. (The informal justification is clear: if a were true, then the implication tells us that b would be true; but we are precisely assuming that b doesn't hold.)

Using this rule we may replace the earlier flawed examples by logically sound deductions:

- S1 • “All professional politicians are corrupt. She is not corrupt, so she can't be a professional politician.”
- S2 • “Whenever I take my umbrella it doesn't rain: since weather.com says it's going to rain, I might as well leave my umbrella at home.”
- S3 • “Since all recent buildings in this area have bad isolation and this room remains cool in spite of the heat outside, the house must be older.”

5.3 SEMISTRIC BOOLEAN OPERATORS

Computer programming fundamentally relies on mathematical logic, to the point that some people consider programming to be just an extension of logic. This is all the more remarkable that modern logic was established in the first few decades of the twentieth century, before there was any hint of computers in today's sense.

Touch of history: The road to modern logic

Logic goes back to the Ancients, Aristotle in particular, who defined the rules of “Rhetorics”, fixing some forms of deduction. In the eighteenth century Leibniz stated that reasoning was just a form of mathematics. In the nineteenth century the English mathematician George Boole defined the calculus of truth values (hence “boolean”). The big push for logic in the following century was the realization that mathematics as practiced until then was shaky and could lead to contradictions; the goal pursued by the creators of modern mathematical logic was to correct this situation by giving mathematics a solid, rigorous basis.

Applying logic to programming brings up some issues often overlooked in purely mathematical uses of logic. An example, important in programming practice, is the need for non-commutative variants of **and** and **or**.

Consider the following question, given a metro line l and an integer n :

“Is the n -th station of line l an exchange?”

We might express it as the boolean-valued expression

```
l.i_th(n).is_exchange
```

[L1]

*Not the correct form
(see [3] below).*

where *is_exchange* is a boolean-valued query of class *METRO_STATION*, indicating whether a station is an exchange; the query *i_th*, seen in the previous chapter, delivers the stations of a line, each identified by an index, here *n*.

← “The stations of a line”, page 61.

The expression above, [1], appears to do the job: *l* denotes a line; *l.i_th(n)*, denotes its *n*-th station, an instance of *METRO_STATION*; so *l.i_th(n).is_exchange*, applying the query *is_exchange* to this station, tells us, through a boolean value, whether it is an exchange station.

But we haven’t said anything about the value of *n*. So *l.i_th(n)* may not be defined since the query *i_th* had a precondition:

← Page 66.

```
i_th(i: INTEGER): METRO_STATION  
-- The i-th station on this line
```

require

```
not_too_small: i >= 1
```

```
not_too_big: i <= count
```

In the absence of further information on *n*, it’s incorrect to use the expression [1] since its result is not defined for $n < 1$ or $n > l.count$.

How can we write a correct expression with the intended meaning? If $n < 1$ or $n > l.count$, it’s reasonable to consider that the answer to our informal question, “Is the *n*-th station of line *l* an exchange?”, cannot be “Yes”, as this would imply that we certify that a certain station is an exchange, and we can’t do this if no such station exists. Since in the boolean world there are only two possibilities, the answer has to be “No!”, meaning formally that the boolean expression should have value **False**. To get this behavior we might try to express the desired property not as [1] but as

```
(n >= 1) and (n <= count) and l.i_th(n).is_exchange
```

[L2]

Still not right (see [3]).

But this is still not good enough. The problem is that if n is out of bounds, for example $n = 0$, the last term $l.i_th(n).is_exchange$ is not defined. If we are only interested in the value of [2], we might not care, because the Conjunction Principle tells us this value can only be **False** since the first term, $n \geq 1$, has value **False**; the second and third terms don't affect the result.

Assume however that the expression appears in a program and gets evaluated during the program's execution. The operator **and**, as we have seen, is commutative; it's legitimate for the execution, when it needs to compute a **and** b , to compute both operands a and b and then combine their values using the truth table for **and**. But then the computation of [2] will fail when it tries to evaluate the last term.

If that evaluation were conceptually required, we could do nothing: a computation that tries to evaluate an expression with undefined value should fail. It's like trying to evaluate the numerical expression $1 / 0$. But in this case we may prefer that when the first term has value **False** the evaluation will, instead of failing, return the value **False**, consistent with the definition of **and**.

We cannot achieve this with the usual commutative boolean operators: we can't prevent their computer versions from evaluating both operands, and then risking failure.

The case illustrated by this example — evaluating a condition that only makes sense if another condition is also satisfied — occurs so frequently in practice that we need a solution. There are three possible ways to go.

The first would be to try to recover from the failure. If an operand to a boolean expression is undefined, so that its evaluation leads to failure, we could have a mechanism that “catches” the failure and tries to see if other terms suffice to determine a value for the expression as a whole. Such a mechanism means that failure is not like real death but more like death in video games, where you can get new lives (as long as you can continue paying). The mechanism exists: it's called **exception handling** and enables you to plan for accidents of execution and try to recover. We'll study it in a later chapter. In the present case, however, it would be (if one dares use the term) overkill. It requires far too much special programming for what is, after all, a simple and common situation.

→ Chapter 11.

The second way would be to decide that **and** as we understand it in programming is not commutative any more (the same would, out of duality, also hold of **or**). In computing a **and** b , we would have the guarantee that b won't be evaluated if a has been evaluated to **False**, the result in that case being **False**. The problem with this approach is that it's unpleasant to make the software version of a well-accepted mathematical concept depart from its mathematical meaning. More pragmatically, the commutativity of **and** and **or** when both operands are defined can help make the computation faster, as it may be advantageous to evaluate the second expression first, or even, if the hardware includes several processors, to evaluate them in parallel.

Such improvement of execution speed, known as **optimization**, is generally not carried out not by programmers but by compilers (the tools that translate your programs to machine code).

The third way — the one we retain — is to accept the usefulness of non-commutative boolean operators but give them different names to avoid any semantic confusion. The new variant of **and** will be written **and then**; by duality we also have a variant of **or**, called **or else**. In each case it's a double keyword, written with a space between the two constituent words. The semantics follows from the previous discussion:

Definitions: Semistrict boolean operators

Consider two expressions a and b which may be *defined* or not, and if defined have boolean values. Then

- a **and then** b has the same value as a **and** b if both a and b are defined, and in addition has value **False** whenever a has value **False**.
- a **or else** b has the same value as a **or** b if both a and b are defined, and in addition has value **True** whenever a has value **True**.

If you are wondering about the name: we say that an operator is *strict* (as in “My mother is *strict* about having everyone at the table before any of us starts eating”) if it insists, to produce its result, on having all operand values available, even those that the evaluation may turn out not to need. An operator is “*non-strict* on an operand if it may in some cases yield a meaningful result even that operand doesn't have a defined value. We call **and then** and **or else** *semistrict* because they are strict on their first operand but not on the second.

Saying “non-commutative” would be acceptable for the operators seen so far, but we'll need semistrict variants of operators such as **implies**, which is not commutative in the first place.

Another way to define the semantics of the semistrict operators is to introduce a variant of truth tables where every operand and result may have three values rather than just two: **True**, **False** and *Undefined*.

← This is the subject of exercise [5-E.13, page 108](#).

Whenever ***a*** **and** ***b*** is defined, ***a*** **and then** ***b*** is defined and has the same value, but the converse is not true. The same holds for **or else** relative to **or**.

With this notation the correct way to express our example condition is

`((n >= 1) and (n <= count)) and then l.i_th (n).is_exchange [L3]`

Version [2] had two **and** operators, but only the second one needs to be turned into an **and then**; between the first two terms, grouped here in parentheses for clarity, a plain **and** is good enough since both will always be defined. This is a general advice:

Touch of Methodology:

Choosing between ordinary and semistrict boolean operators

In expressing contracts and other conditions:

- Use the ordinary boolean operators, **or** and **and**, when you can guarantee that both operands are defined whenever the execution needs to evaluate the condition.
- If a condition only makes sense when another is false, use **or else**.
- If a condition only makes sense when another is true, use **and then**.

Our example, [3], corresponds to the last case.

In the first case it wouldn't be *wrong* to use the semistrict version, but this would needlessly prescribe a particular evaluation order; it's preferable to avoid such "overspecification" and stick instead to the operators with standard mathematical properties. This also leaves compilers free to optimize the order of operand evaluation.

The notion of semistrict operator is applicable to more than mathematical logic and software:

Touch of Practice:

Semistrict operators and you

The semistrict operators reflect modes of reasoning that are common in daily life.

Wherever you see the phrase "*if any*" you may suspect that semistrictness is involved. A credit application form might stipulate that the spouse, *if any*, must be a co-signer; we may understand this as *is_single or else spouse_must_sign* or, in more programming-oriented terms:

`(spouse = Void) or else spouse.must_sign`

where *Void* denotes the absence of an object. In either form the second operand of the **or else** would not make sense with a strict **or**, since when the first operand has value **False** the notion of spouse is not defined.

→ Studied in "[VOID REFERENCES](#)", 6.3, page 113.

Semistrict implication

Implication also has a semistrict variant. A routine with arguments l : *METRO_LINE* and i : *INTEGER* might use the precondition

```
((i >= 1) and (i <= count)) implies l.i_th (i).is_exchange
```

meaning: apply the routine only if the i -th station of line l , assuming it exists, is an exchange.

This makes sense only with a semistrict interpretation of **implies**. Such a scheme — an expression of the form a **implies** b where b is defined only when a is true — occurs so frequently that for this operator, which is not commutative in the first place, the semistrict version seems appropriate in all cases. Such a convention is also consistent with the Implication Principle and its insistence that a **implies** b yields **True**, regardless of b , whenever a has value **False**.

So we take the semistrict version as the definition of **implies**:

Definition: Implication with possible undefinedness

The value of a **implies** b , for any a and b where b may not be defined, is the value of

(not a) or else b

The “ordinary life” example of semistrictness cited above falls in this category; we may now write it with semistrict **implies** as

```
(spouse /= Void) implies spouse.must_sign
```

Many uses of “if any”, for example in legal texts, follow this form.

5.4 PREDICATE CALCULUS

The concepts discussed so far in this chapter belong to a part of logic called **propositional calculus**, meaning that it deals with basic *propositions*, each stating a single property p that might be true or false: n has a positive value, I am the Pope, it's full moon tonight. “Single property”, in these examples, means that p characterizes a single object — the number n , me, the current night — or a finite set of explicitly listed objects, as in “I am not the Pope and it's not a full moon tonight”.

Another theory is directly useful in programs and discussions of programs: **predicate calculus**, which considers whether a property holds for the elements, not necessarily specified individually, of a *set* of objects.

Generalizing “or” and “and”.

Given a set of objects E and a property p of objects, predicate calculus deals with two basic questions, generalizing “or” and “and”:

- 1 • Does *at least one* of the objects in E satisfy p ?
- 2 • Does *every one* of the objects in E satisfy p ?

For example, we have seen that a Metro line contains stations, and that some stations are exchanges (they belong to two or more lines). We may ask, taking an arbitrary line as example:

- L1 • Is at least one of the stations of Line 8 an exchange?
- L2 • Are all of the stations of Line 8 exchanges?

If you know all the stations by name you can express these questions as boolean expressions. L1 is an **or** expression and L2 is an **and** expression:

L1 • `Station_Balard.is_exchange or Station_Lourmel.is_exchange or Station_Boucicaut.is_exchange or ... [Include all stations on line] ...`

L2 • `Station_Balard.is_exchange and Station_Lourmel.is_exchange and Station_Boucicaut.is_exchange and ... [Include all stations on line] ...`

using the boolean-valued query `is_exchange` of class `METRO_STATION` to tell us if a station is an exchange. You would have to complete the expressions by including a term for each station of the line.

You can avoid naming the line's stations by using the query *i_th* of class *METRO_LINE* which, as seen in the preceding chapter, gives us the *i*-th station of a line for any applicable *i*:

I1 • *Line8.i_th (1).is_exchange or Line8.i_th (2).is_exchange or*
 ... [Include all values from 1 to *Line8.count*]

I2 • *Line8.i_th (1).is_exchange and Line8.i_th (2).is_exchange and*
 ... [Include all values from 1 to *Line8.count*]

but that is still inconvenient as you must explicitly list all stations. In particular you can't write, for either question, an expression that would make sense for *any* line, since different lines have different numbers of stations.

Predicate calculus addresses such cases by introducing **quantifier** expressions that describe the application of a property to a set of objects, letting you specify only that set, for example a Metro Line, rather than every object individually — every station. There are two quantifiers:

- The **existential quantifier**, *exists*, or \exists in mathematical notation, stating that *at least one* member of the set satisfies the property.
- The **universal quantifier**, “*for_all*”, or \forall in mathematical notation, stating that *every* member of the set satisfies the property.

When you would need boolean operations on an arbitrary number of operands, *exists* generalizes **or**, and *for_all* generalizes **and**. If *Stations8* denotes a list of stations, the mathematical notations are:

Q1 • $\exists s: \textit{Stations8} \mid s.is_exchange$

Q2 • $\forall s: \textit{Stations8} \mid s.is_exchange$

which you may read aloud respectively as:

- **There exists** an *s* in *Stations8* such that *s.is_exchange* is true.
- **For all** *s* in *Stations8*, then *s.is_exchange* is true.

Rather than using a bar “|” as above to separate the property, here *s.is_exchange*, from the specification of the set of objects across which it will range, mathematicians often use a period “.” or a comma “,”; but for us this would be ambiguous since, as you know, we need these symbols for other purposes.

Q1 and Q2 are mathematical notations, not programming notations. We'll see shortly how to express such properties in a program.

Precise definition: existentially quantified expression

The notations using existential and universal quantifiers, as just illustrated, are new forms of boolean expression, complementing the expressions of propositional calculus seen earlier in this chapter.

The definition of the existential quantifier is straightforward:

Definition: Existentially quantified expression

The value of the expression

$\exists s: \text{SOME_SET} \mid s.\text{some_property}$

is **True** if and only if at least one member of the given set *SOME_SET* satisfies the given property *some_property*.

For example let X be the set of integers $\{3, 7, 9, 11, 13, 15\}$ (that is to say, the set consisting of the integers listed) and for any integer n let $n.\text{is_odd}$ be the property that n is odd, $n.\text{is_even}$ the property that it's even, and $n.\text{is_prime}$ the property that it is a prime number. Then:

- $\exists n: X \mid n.\text{is_odd}$ means that at least one member of X is odd; the expression has value **True** since we can take, for example, 3 as evidence that there is one such member. In this case we may take any other member of the set as evidence since they are all odd.
- $\exists n: X \mid n.\text{is_prime}$ means that at least one member of X is prime; this expression also has value **True** since we may take, for example, 3 again as evidence. It doesn't matter that some other member or members, such as 9, don't satisfy the property since the truth of an existentially quantified expression only requires one example.
- $\exists n: X \mid n.\text{is_even}$ means that at least one member of X is even; this expression has value **False** since no element of X is even.

These examples illustrate how you may prove or disprove an existentially quantified expression $\exists s: \text{SOME_SET} \mid s.\text{some_property}$:

- E1 • To prove that it is true, it suffices to exhibit *one* element of *SOME_SET* that satisfies the property. Once you have found such an element, others have no influence on the result. This means in particular that you may not need to investigate all elements of the set.
- E2 • To prove that it is false, you must prove that *no* element of *SOME_SET* satisfies the property. That *some* don't satisfy it is not enough to determine the result. This means in particular that you **must** consider all the elements.

Precise definition: universally quantified expression

For an expression using a universal quantifier

$$\forall s: \text{SOME_SET} \mid s.\text{some_property}$$

the informal definition of its value is that it's **True** if and only if every element of *SOME_SET* satisfies *some_property*. This is not quite precise enough, however, for reasons having to do with the case of an empty set. A better approach is base the definition on what has just been specified for *existentially* quantified expressions:

Definition: Universally quantified expression

The value of the expression

$$\forall s: \text{SOME_SET} \mid s.\text{some_property}$$

is the value of

$$\text{not } (\exists s: \text{SOME_SET} \mid \text{not } s.\text{some_property})$$

This says that the \forall expression has value **True** if and only if there is *no* member of the given set that does *not* satisfy the given property. It sounds like a rather contorted way of expressing what we want: that every element satisfies the property. In your writing classes you were probably told to avoid double negation, replacing “There’s no course I don’t like in this great university!” by “I like all courses here”. The reason for the double negation here is that we must be careful about the case of *empty* sets. Before examining this case, let’s consider again our example set of integers *X* made of the elements 3, 7, 911, 13 and 15, and those only:

- $\forall n: X \mid n.\text{is_odd}$ means that all members of *X* are odd; the expression has value **True** since 3, 7, 9, 11, 13 and 15 are all odd numbers.
- $\forall n: X \mid n.\text{is_prime}$ means that all members of *x* are prime numbers; this expression has value **False** since we can take, for example, 9 as evidence that at least one member is not prime. We could also use another non-prime member as evidence — the other possibility is 15 — but one is enough to prove that the universally quantified expression is false.
- $\forall n: X \mid n.\text{is_even}$ means that all members of *X* are even; this expression has value **False** since, for example, 3 is not even. Here *any* other member of the set could serve as evidence since none is even, but again one is enough.

These examples illustrate how you may prove or disprove a universally quantified expression $\forall s: \text{SOME_SET} \mid s.\text{some_property}$ (compare with those for existential quantification, [E2](#) and [E1](#) above):

- U1 • To prove that it is true, you must prove that *every* element of SOME_SET , if any, *satisfies* the property. That *some* satisfy it is not enough to determine the result. This means in particular that you **must** consider all the elements.
- U2 • To prove that it is false, it suffices to exhibit *one* element of SOME_SET that *does not satisfy* the property. Once you have found such an element, others have no influence on the result. This means in particular that you may not need to investigate all elements of the set.

The relationship between the existential and universal quantifiers generalizes the duality between **or** and **and**. In particular the following two properties generalize [De Morgan's Laws](#):

← Page [83](#).

$\text{not } (\exists s: E \mid P) = \forall s: E \mid \text{not } P$ $\text{not } (\forall s: E \mid P) = \exists s: E \mid \text{not } P$

The first property follows from the definition of \forall ; the second property follows from applying the first to **not** P and negating both sides.

The case of empty sets

The set SOME_SET of possible values considered in a quantified expression might be empty. The effect on the two quantifiers reflects their duality:

- $\exists s: \text{SOME_SET} \mid s.\text{some_property}$ is true, according to its definition, if and only if some member of SOME_SET satisfies *some_property*. If SOME_SET is empty, it has no member, and hence no member satisfying the property. So the value of the expression in this case is **False**.
- $\forall s: \text{SOME_SET} \mid s.\text{some_property}$ is false if and only if some member of SOME_SET does *not* satisfy *some_property*. If SOME_SET is empty, there won't be any such "counter-example" member since there is no member at all. So the value of the expression in this case is **True**.

The second case can also be seen as a result of the [definition](#) of the universal quantified expression $\forall s: \text{SOME_SET} \mid s.\text{some_property}$ in terms of the existentially quantified one, as

← Page [100](#).

$\text{not } (\exists s: \text{SOME_SET} \mid \text{not } s.\text{some_property})$
--

By the previous convention, the whole $(\exists s: \text{SOME_SET} \mid \dots)$ expression in parentheses has value **False** if *SOME_SET* is empty, so the \forall expression, deduced from it by applying **not**, has value **True**.

Concretely, this simply means that we may consider every statement of the form “Every object of such-and-such a kind satisfies this property” as true if there is no object of the given kind. So I can say “I promise to you that every blond student in this room will be elected Pope before the end of the year”, and even back this with “if not, I’ll pay every one of you a million euros on January 1st”, if I have (carefully) checked that everyone in the room has black hair. The statement “Every blond student in this room will be elected Pope” is indeed true because it is of the form $\forall s: \text{SOME_SET} \mid \dots$ for an empty *SOME_SET*, which is true regarding of what comes after the “|”.

Having studied logic, however, you should never promise anything like “A blond student in this room will be elected Pope” because it makes you responsible for identifying a fair-haired student *and* rigging the election.

As a result of these observations, the official name of the universal quantifier, “*For all*”, is not so good because “*all*” suggests, at least informally, that there are some elements to be talked about. Better names would be “For all, if any”, or just “For any”. They wouldn’t absolutely preclude confusion anyway, so we’ll continue saying “For all” like everyone else, but you have to remember that this is just an informal name and that the mathematical interpretation gives a **True** answer if there are no elements to be probed for the property. For “*Exists*”, the answer in this case is **False**.

Another way to express this property is that if we consider an existential quantification on a set of values to mean a_1 **or** a_2 **or** ... **or** a_n , and the universal quantification to mean a_1 **and** a_2 **and** ... **and** a_n , then as n goes to zero the disjunction will yield false and the conjunction will yield true. This is in line with earlier observations that a **or** b is true if and only if at least one of a and b is true, and a **and** b is false if and only if at least one of a and b is false.

Yet another informal interpretation relates this property to the earlier discussion of how “implies” always yields **True** when the antecedent is **False**. We might understand $\forall s: \text{SOME_SET} \mid s.\text{some_property}$ as a way of saying that “ s is a member of *SOME_SET*” implies $s.\text{some_property}$. If *SOME_SET* is empty the antecedent is **False** for every possible x , so the implication is true.

5.5 FURTHER READING

The material in this chapter is introductory; as part of a computer science curriculum you will most likely take a course specifically devoted to logic. A standard textbook on the topic, which requires a solid background in general mathematics but defines all the concepts it uses, is

Elliot Mendelson: *Introduction to Mathematical Logic*, fourth edition, Chapman & Hall/CRC, 1997.

5.6 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Logic provides the techniques for reasoning in a precise and rigorous ways. It provides the basis of both mathematics and programming.
- *Propositional calculus* defines operations on “boolean variables” that can take either of the values **True** and **False**. The basic “boolean operations” are negation (**not**), non-exclusive disjunction (**or**) and conjunction (**and**).
- Disjunction and conjunction are “dual” of each other: replacing either of them by the other one, negating the operands and negating the result yields a property of the other. This is expressed in particular by “De Morgan’s Laws”.
- Disjunction and conjunction can be generalized to any number of operands through the quantifiers \exists and \forall of *predicate calculus*, which apply to the members of a given set.
- Implication can be defined simply in terms of disjunction: *a implies b* is the same as **(not a) or b**. Implication can be used to deduce new properties from previously proven ones; it does not connote causality.
- In their application to programming, the boolean operations have *semistrict* versions that yield a value even in some cases for which the second operand is not defined. The semistrict variants of **or** and **and** are **or else** and **and then**; **implies** is defined as semistrict.

New vocabulary

Antecedent	Boolean value	Boolean expression
Boolean operator	Boolean variable	Conjunction
Consequent	Contradiction	Disjunction
Existential quantifier	Implication	Logic
Negation	Opposite	Predicate calculus
Propositional calculus	Quantifier	Satisfiable
Satisfies	Strict	Stronger
Tautology	Truth assignment	Truth table
Universal quantifier	Weaker	

5-E EXERCISES

5-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

5-E.2 Properties of boolean operators

(Prove your answers.)

- 1 • Is **and** reflexive?
- 2 • Is **or** reflexive?
- 3 • Is equivalence associative?

5-E.3 Twisted logic

“Whenever the temperature in the city raises above 30 degrees a pollution alert will result, so because the temperature today is only 28 degrees there won’t be a pollution alert.”

- 1 • Informally, what’s wrong with this statement?
- 2 • Introducing the appropriate boolean variables, express this statement as a boolean expression.
- 3 • Prove that it is not a tautology. (*Hint*: give a set of variable assignments that makes it false).
- 4 • Is it a contradiction? (Prove your answer.)

5-E.4 Appropriate warning?

A sign at the entrance to a computer center once read: “*Entrance is prohibited to people who are not authorized or accompanied.*” We accept that there is no ambiguity as to what “authorized” means (someone who been granted the appropriate credentials), and that “*accompanied*” means “accompanied by an authorized person”.

- 1 • Introducing appropriate boolean variables, express this rule as a boolean expression.
- 2 • Explain why the expression doesn’t capture the interdiction that the sign’s author probably intended. (*Hint*: Use De Morgan’s Laws.)
- 3 • Write the expression reflecting the rule that was most likely intended.
- 4 • Using this expression as a guide, propose an improved rewrite of the English text for the sign.

5-E.5 Inequality

Write the truth table for the inequality operator \neq .

← Introduced on page [83](#).

5-E.6 Associativity and implication

Is the **implies** operator associative? (Prove your answer.)

5-E.7 Police logic

Are Thomson and Thompson, the two policemen in the Tintin extract, justified in accepting Captain Haddock’s final explanation? ← Page [89](#).

5-E.8 Implication and negation

The discussion of implication noted that

← Page [90](#).

$$(a \text{ implies } b) = ((\text{not } a) \text{ implies } (\text{not } b))$$

is not a tautology. By simplifying this expression — through theorems introduced in this chapter, not truth tables — show under conditions (e.g. which truth assignments) it holds.

5-E.9 Implication

- 1 • Prove that for any boolean expressions a and b the following is a tautology:

$((a \text{ implies } b) \text{ and } ((\text{not } a) \text{ implies } b)) \text{ implies } b$

- 2 • The sign shown on the right, spotted in Zurich near the ETH, reads: “Reasonable drivers don’t park here. For others, it’s forbidden!”. Using appropriate boolean variables, including $is_reasonable$, $park_here$, $parking_prohibited$, express this injunction as one boolean expression.
- 3 • Prove that if this expression is a tautology, and drivers obey parking prohibitions, then $park_here$ is a contradiction.



5-E.10 “Exclusive or” as a germ of all things boolean

“Exclusive or”, written **xor**, is a boolean operator of two operands such that $a \text{ xor } b$ is true if and only if either a or b , but not both, is true. We may state this property by defining $a \text{ xor } b$ as

$(a \text{ or } b) \text{ and } (\text{not } (a \text{ and } b))$

[L4]

- 1 • Write the truth table for **xor**.
- 2 • If a is a boolean variable, what is the value of $a \text{ xor } a$? (Prove your answer from either the definition or the truth table.)
- 3 • Prove that $a \text{ xor } b$ always has the same value as $\text{not } (a = b)$

For each of the following boolean expressions (with zero, one or two operands), give another boolean expression that for any value of the operands yields the same value as the given expression, and involves nothing else than the operands, **True** and **xor** (in particular, no other operator); prove your answers.

- 4 • **False**
- 5 • **not a**
- 6 • $a = b$
- 7 • $a \text{ and } b$
- 8 • $a \text{ or } b$
- 9 • $a \text{ implies } b$

(The existence of an **xor** equivalent for every boolean operation makes **xor** a particular interesting operator, holding the germ of all others. Designers of electronic circuits based on boolean logic have taken advantage of this property.)

5-E.11 Properties of “exclusive or”

Based on the above definition [4] of **xor**, the “exclusive or” operator, prove or disprove the following properties :

- 1 • **xor** is commutative.
- 2 • **xor** is associative.
- 3 • $x \text{ xor } (a \text{ xor } x) = x$ for any a and x .

5-E.12 The blue hats and the red hats

A hundred persons are standing in line, each wearing a hat that is either blue or red. They can each see the hat colors of those ahead in the line, but neither their own nor those of people behind.

Starting with the back of the line — the person who sees all others — they will each, in turn, shout a color name, “**Red!**” or “**Blue!**”, which all can hear.

You are asked to devise a strategy, which they will all adopt beforehand, to maximize the number of people who — regardless of the distribution of hat colors, about which you know nothing — are guaranteed to shout the color of their own hats.

Noting the following properties will help:

- A simple strategy is for the first person, the third, the fifth and so on to shout the color of the person immediately ahead (the second, the fourth, the sixth and so on), who then repeats that color, guaranteed to be correct. This gives a lower bound: a good strategy should guarantee *at least* 50 correct results.
- No strategy can guarantee that the first shouter, who doesn’t see his or her color, will be correct. This gives an upper bound: a strategy can guarantee *at most* 99 correct results. We may restate the problem as asking how close you can get to this theoretical maximum.
- There is nothing probabilistic about the problem. Even if we had some information about the distribution of colors, it wouldn’t help since the strategy must maximize the number of answers *guaranteed* correct, not some probability of correct answers.

Hint: the preceding exercise helps.

5-E.13 Truth tables with undefinedness: semistrict boolean operators

Assume an extension of propositional calculus with three values instead of two: **True**, **False**, and *Undefined*. For example *l.i_th (i)* has value *Undefined* if *i* doesn't satisfy the precondition of *i_th*.

Considering that each of *a*, *b* and the resulting expression may take on any of these three values, write the truth tables (each with nine entries) for:

- 1 • *a or else b*
- 2 • *a and then b*

5-E.14 Truth tables with undefinedness: ordinary boolean operators

As in the preceding exercise, assume that boolean values include **True**, **False**, and *Undefined*. Explaining the reason for your answer, propose truth tables for:

- 1 • *a or b*
- 2 • *a and b*

(Here more than one set of truth tables may make sense, so what's interesting is how you justify your proposed solution.)

6

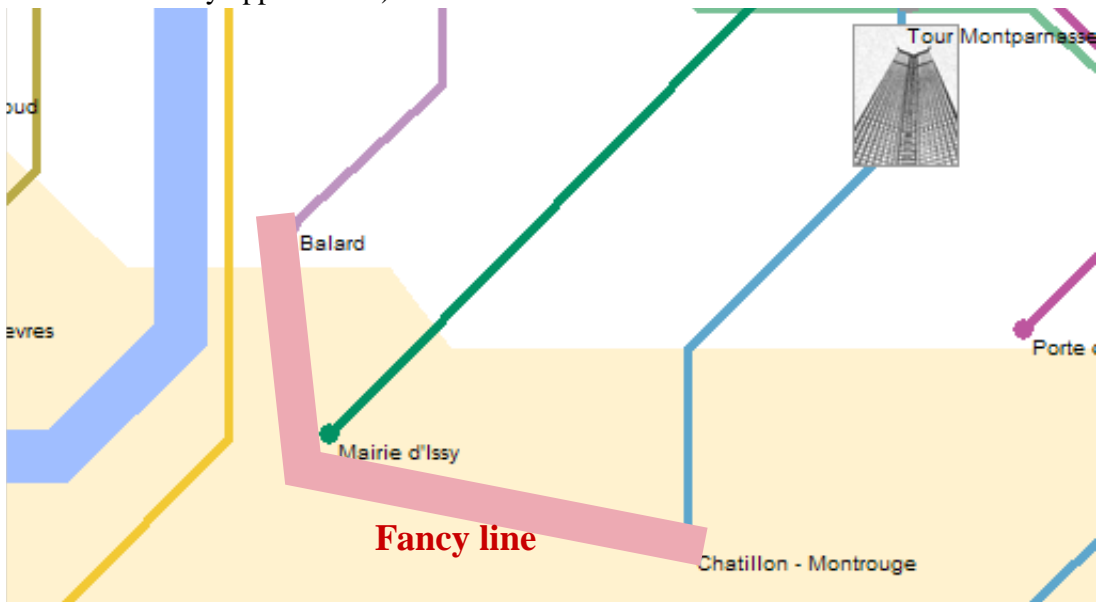
Creating objects and executing systems

After our excursion into the mathematical foundation we are back to the techniques of programming.

In earlier examples we have used names such as *Paris*, and *Routel* to access objects that someone else creates for us — mysteriously so far. It's time to see how we can, in our own programs, create our own objects ourselves.

We'll create a fictitious line of the metro, *fancy_line*, connecting some real stations. Contrary to our previous examples such as *Line8*, the line *fancy_line* is not predefined; we have to build it ourselves. In doing so we'll have to create other objects, such as those representing stops on the line.

Our *fancy_line* will connect three stations as shown (people living in the area would really appreciate it):



6.1 OVERALL SETUP

Our system for this chapter is called *Fancy*. Bring it up now, using the same techniques as in previous chapters. Bring up the class of interest for the present discussion, *LINE_BUILDING*, which initially looks just like this:

```
class LINE_BUILDING inherit
  TOUR
  feature
    build_a_line is
      -- Build an imaginary line and highlight it on the map.
    do
      Paris.display
      Metro.highlight
      -- "Create new line and fill in its stations"
      fancy_line.illuminate
    end
end
```

The line -- “Create new line and fill in its stations” starts with two hyphens and hence is a comment, but of a special kind known as **pseudocode**, meaning that it stands for actual program text (also known as “code”) which we intend to fill in later, as we develop the program:

← Page 43.

Definition: Pseudocode

Pseudocode is informal text standing for program elements to be added later.

We will use pseudocode (rather than a margin note such as “*The part you’ll fill in*”) to give an informal English description of program elements that we are not expressing as actual program text yet, either because we can’t or because this would force us to go into a specific aspect of the program and lose track of the big picture.

← Like for example on page 18.

This technique will become essential as we start writing more complex software. It is part of *top-down design* discussed in a later chapter.

Pseudocode will use the convention illustrated by the example:

Touch of Style: Highlighting pseudocode

Write pseudocode elements as comments, with their text enclosed in quotes and (if color is available) appearing in red.

By using comments for pseudocode you ensure that your program, even if not complete, is syntactically correct; it may not be interesting yet to execute it, but you can compile it, so that the compiler will find any errors that you let slip through, such as incorrect use of types. It's a basic methodological rule that programs should be compilable at all stages of their development.

Marking these comments in a special way (quotes and, in printed text, color) reminds you that they are not just ordinary comments annotating existing code, but placeholders for code that you must add at some point.

6.2 ENTITIES AND OBJECTS

The first thing we need in our class is a feature representing the line to be built. We call it *fancy_line*. This is also an opportunity to make the pseudocode comment more precise, part of the top-down development process:

```

class LINE_BUILDING inherit
  TOUR
  feature
    build_a_line is
      -- Build an imaginary line and highlight it on the map.
    do
      Paris.display
      Metro.highlight
      -- "Create fancy_line and fill in its stations"
      fancy_line.illuminate
    end

    fancy_line: METRO_LINE
      -- An imaginary line of the Paris Metro
  end
end

```

In **pseudocode**, any actual program elements, such as *fancy_line*, will appear in their usual blue to signal that there's nothing "pseudo" about them.

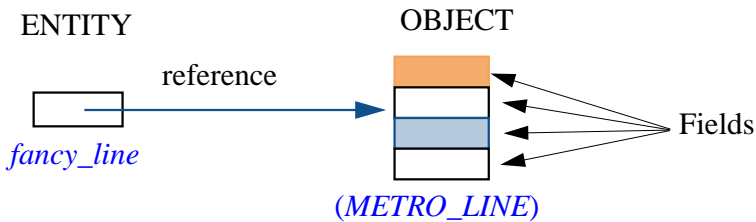
Once the procedure *build_a_line* has been executed, *fancy_line* will denote an object representing a Metro line. Within the class text, the **identifier** *fancy_line* will indeed denote, at run time, an instance of class *METRO_LINE*.

← The notion of identifier was introduced on page 47.

Identifiers may denote many things: they can be names of classes, like *METRO_STATION*, or of features, like *i.th*. An identifier such as *fancy_line* whose role is to denote run-time values, such as objects, is called an **entity**. (You will also encounter the term *variable*, as in mathematics; we'll use it too, for entities whose value may change, but it's too restrictive here since some of our entities we need actually have *constant* values.) In this case the entity *fancy_line* is the name of a feature, but we'll encounter other kinds of entity.

If, at some instant of the execution, the value of an entity represents an object, we say that the entity is **attached** to the object.

The following picture helps visualize the notion of entity and attached run-time object:



During execution: entity and attached object

This shows the relationship: the entity is a name in the program and at run time it will denote, through a “reference”, an object in memory. The notion of reference expresses the association and will be defined more precisely in a later chapter. The object, as defined earlier, is a collection of data; it is made more precisely, as suggested by the picture, of a set of **fields** each holding a data unit (for example an integer or boolean value). The data that our programs manipulate during execution is *entirely* made of such objects, each with its fields. The fields of a *METRO_STATION* object might, for example, include the station’s coordinates on the map, the name of the station etc.

→ “REFERENCE ASSIGNMENT”, 9.5, page 234 47.

← Definition of “object”: page 33.

Note the conventions in diagrams such as the above giving a snapshot of the object structure, or part of it, during execution:

- An object is represented as a **rectangle**, with its fields represented as sub-rectangles.
- Next to each object, usually below, you’ll see in parentheses the name of its generating class — the class of which it is an instance, here (*METRO_LINE*).

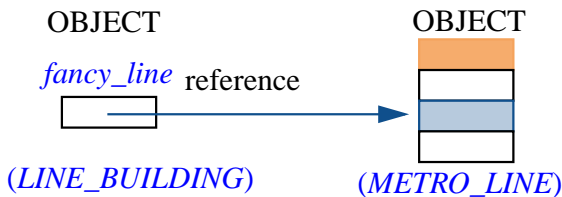
← Definition of “generating class”: page 54.

6.3 VOID REFERENCES

In considering the execution of *build_a_line* and the value of *fancy_line* we must pay particular attention to references and their relation to objects.

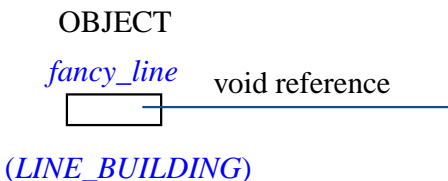
The initial state of a reference

Assume we have an instance of *LINE_BUILDING*. You might think that because the class declares a query *fancy_line* of type *METRO_LINE*, we may always assume that its instance contains a reference to an instance of *METRO_LINE* as suggested above:



Line entity and attached METRO_LINE object

Not so. We do have one object, the one on the left in the figure, an instance of *LINE_BUILDING*, with only one field corresponding to the query *fancy_line*. Let's assume this object has just been created; this is the result of a “creation instruction”, which we will shortly see how to write. The instruction only gives us the *LINE_BUILDING* object. If you need any other, your program will have to create it explicitly. So the true state of program execution after creation of an instance of *LINE_BUILDING* looks like this:



Object structure at the beginning of execution

The field for *fancy_line* contains a reference. But because no instruction has been executed yet to create other objects, that reference is **void**, meaning that it is not attached to any object; the figure shows the convention for void references, reminiscent of the “grounded” symbol in electricity.

This is one of the two possible states for a reference:

Definition: States of a reference

At any time during execution, the value of an entity denoting a reference is one of:

- **Attached** to a certain object.
- **Void**.

The predefined feature *Void* denotes a void reference. So at any time during execution, if *x* denotes a reference, the condition

```
x = Void
```

has value *True* if and only if the value of *x* is a void reference, and

```
x /= Void
```

← */=* is inequality; see page 83.

if and only if it is attached to an object.

The trouble with void references

The basic mechanism of computation was introduced as *feature call*, of the form *x.f*, or *x.f(a, ...)* with arguments. This applies feature *f* to the object to which *x* is attached. But now with void references we have the possibility that, at some time during execution, if *x = Void* holds, the reference that *x* denotes is not attached to any object. The feature call is erroneous in that case.

← “*Dissecting the program*”, page 25.

To see the effect of such a bug, try to execute the system as it stands, leaving the line of pseudocode as a comment:

```
class LINE_BUILDING inherit
  TOUR
feature
  build_a_line is
    -- Build an imaginary line and highlight it on the map.
    do
      Paris.display
      Metro.highlight
      -- The next line should have been replaced by code!
      -- “Create fancy_line and fill in its stations”
      fancy_line.illuminate
    end
  fancy_line: METRO_LINE
end
```

After the initial calls (*Paris.display* and *Metro.highlight*) execution stops abruptly, displaying a message in EiffelStudio stating that an **exception** has occurred:

*Abnormal
termination and
the resulting
message*

An exception is an abnormal event occurring during program execution. Sometimes you can plan for exceptions and write program element that will try to recover, using techniques seen in a later chapter; otherwise, an exception will just cause erroneous program termination, or **failure**, as happens here.

Another example of exception, “arithmetic underflow”, is an attempt to compute the division a / b where b has value zero, or a non-zero value that’s too small for the computer’s number representation system.. Every kind of exception has a name, such as “*Arithmetic underflow*” or, “*Feature call on void reference*”, which appears in the EiffelStudio failure message.

The EiffelStudio message indicates what the exception was here: “Feature call on void reference”.

Not every declaration should create an object

To avoid the exception in our last example, we may change the creation procedure *build_a_line* so that before the call *fancy_line.illuminate* it will have created an object and attached it to *fancy_line*. We’ll do this shortly. You may, however, question the behavior. Why have void references at all and hence create the resulting risk of void call exceptions? In other words shouldn’t we be able to assume that a declaration such as

```
fancy_line: METRO_LINE
```

will at run time have the effect of creating an object — an instance of *METRO_LINE* — and attach it to *fancy_line*?

The answer is no. Several reasons justify the convention that references are initialized to void, and that you get objects only by creating them explicitly through your program.

The basic reason is that some objects simply don't exist. That's true in the non-software world: a person *may* have a spouse, but not everyone is married. So it should also be true in software that models that world: in a class *PERSON*, appearing for example in tax management software, you may want to include a feature

```
spouse: PERSON
```

for which the possibility of a void reference is essential: it will represent the case of an unmarried person. Even if we assumed everyone is married, it would still make no sense to create an object for *spouse* every time we create an object of type *PERSON*: then it too would have its *spouse* reference, for which we would have to create another instance of the class, starting an infinite chain. So the reasonable solution is to initialize the field to a void reference, and let the program create an object when appropriate.

Let's look at the example a little more in depth. When a person does have a *spouse*, there is a constraint: the spouse is also married, and has, as a spouse, the original person. A picture shows this better than words:



Monogamy

and a formula says it even better than a picture: the invariant of class *SPOUSE* should have a clause that reads

```
(spouse != Void) implies (spouse.spouse = Current)
```

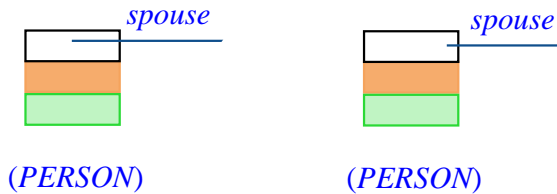
Current, used in relation to an object, denotes the object itself. The clause says that if a person has a spouse, then that spouse's spouse is the original person. → More about *Current* in ...

As we'll see, *Current* is never void since it denotes an object. So from this invariant clause we may deduce another: *(spouse != Void) implies (spouse.spouse != Void)*: if you are a married person, your spouse is married too. Don't underestimate the benefit of expressing such seeming banalities: software development involves clarifying the intuitive knowledge that we may have about a problem domain, and then formalizing it using the tools of logic, for example in class invariants.

Another observation on the above invariant clause: if you have carefully followed the [discussion](#) of non-commutative boolean operators, you will have noticed that this clause requires the non-commutative version of **implies**, since the second operand wouldn't be defined if *spouse* is void.

← “*Semistrict implication*”, page 96.

This shows further why we shouldn't jump to create an object every time there's an entity declaration. Both objects on the preceding figure should probably start their lives celibate, with void *spouse* references:



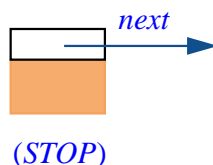
Double celibacy

Later on some instructions will attach the *spouse* reference of the first object to the second and conversely, yielding the state shown on the earlier figure. Such *reattachment* instructions don't create any new objects; they simply attach references to existing objects. We'll study them in a later chapter.

The role of void references

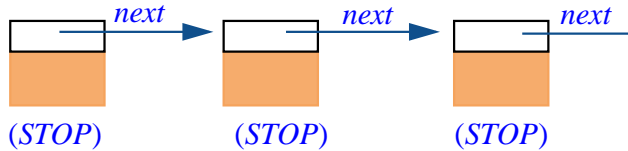
Consider a reference appearing in a field of an object, such as the *spouse* field of a person object. If itself attached to an object, it indicates the presence of certain information, represented by that object. If it's void, it indicates that such information doesn't exist. This is particularly useful when we use references to *link* objects in a more complex structure.

For example we might decide to represent a metro line (any instance of class *METRO_LINE*) by one or more instances of a class *STOP*, representing particular stops on the route. One possible technique (we'll see many others) is to have, in every instance of *STOP*, a field *next* indicating the next stop on the route. So an instance of *STOP* will look like this:



*A stop
(provisional)*

where the solid part represents fields providing other information on the stop. Then a full line will be a set of such objects, each but the last linked to the next one by a *next* reference:



A linked line

Note how the last object uses a void reference for *next* to indicate that there is no *next* object in this case. Terminating such structures is one of the principal uses for void references.

6.4 CREATING SIMPLE OBJECTS

I hope you haven't lost track of our goal in this chapter, which is to create our *fancy_line* as pictured at the very beginning, with three stations. We're almost there but first we need to create the objects representing the stops on the line.

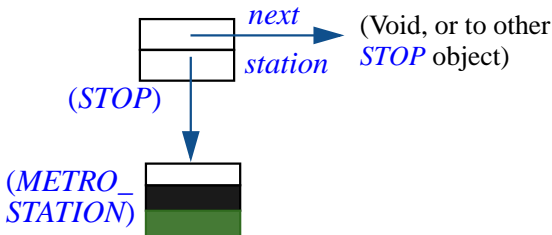
These auxiliary objects will be instances of the class *STOP* just mentioned. Can you see why we need this class, and not just instances of class *METRO_STATION*?

The last figure gives a clue. A stop on a line is associated with a station, but it's a different object because it represents the station *as belonging to the line*. A query such as "What is the next station?" is not a feature of the station; it's a feature of the station as belonging to the line. That's because some stations (exchanges) belong to two or more lines. On the following figure, the "next" station for Gambetta (going as usual from South and West to North and East) depends on which of its two lines you take.



*More than one
"next" station*

A *STOP* object will be very simple: it will contain a reference to a station, and a reference to the next object. We'll require the station reference to be always non-void; the *next* reference may be void at the end of a line.



A stop (final)

Although with what we are learning now we would soon be able to create the station objects themselves, we won't need to; they are available from class *TOUR* under the names *Station_Montrouge*, *Station_Issy*, *Station_Balard*. So we only worry about *STOP* objects here.

A first version of class *STOP* called *SIMPLE_STOP* has the following interface (bring it up under EiffelStudio):

```

class SIMPLE_STOP feature

    station: METRO_STATION
        -- Station which this stop represents

    next: SIMPLE_STOP
        -- Next stop on same line

    set_station (s: METRO_STATION)
        -- Associate this stop with s.
    require
        station_exists: s /= Void
    ensure
        station_set: station = s

    link (s: SIMPLE_STOP)
        -- Make s the next stop on the line.
    ensure
        next_set: next = s

-- This really calls for an invariant stating station /= Void; it will
-- be added to the next version of this class, called just STOP.
end

```

The query *station* yields the associated station; the query *next* yields the next stop. For each of these we have a command: *set_station* to associate the stop to a certain station, and *link* to link it to another stop on the same line.

Here is how to create an instance of this class. Assume that (along with *fancy_line: METRO_LINE*) we have declared

```
stop1: SIMPLE_STOP
```

Then in procedure *build_a_line* we may create a stop:

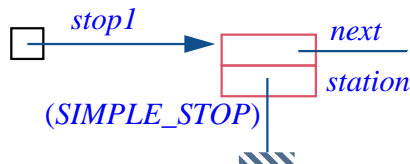
```
build_a_line is
  -- Build an imaginary line and highlight it on the map.
  do
    Paris.display
    Metro.highlight
    create stop1
    -- "Create more stops and finish building fancy_line"
    fancy_line.illuminate
  end
```

The instruction *create stop1* is a **creation instruction**. It's the basic operation that produces objects at run time. Its effect is exactly as the keyword **create** suggests: create an object, and attach the listed entity, here *stop1*, to that new object. In pictures: starting from a state in which *stop1* is void



Before creation instruction

executing *create stop1* attaches it to an object created for this purpose:



After creation instruction

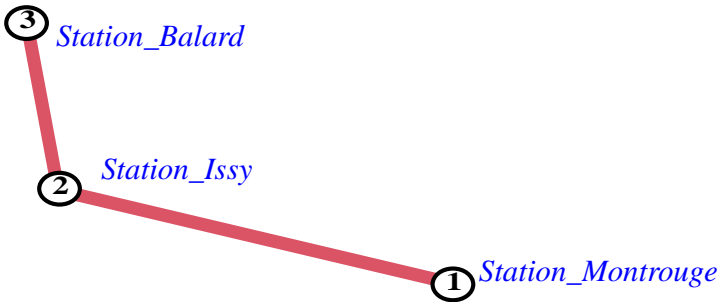
The **create** instruction doesn't need to specify the type of object to be created, since every entity such as *stop1* is declared with a type, here with the declaration *stop1: SIMPLE_STOP*. The type of the object to be created is the type declared for the corresponding entity, here *SIMPLE_STOP*.

As a consequence of the the earlier discussion, all reference fields of the new object are set to *Void*. We can attach them to actual objects using the commands *set_station* and *link*. This enables us to build all the stops of *fancy_line* (the *METRO_LINE* object itself will follow). We declare the three stops:

```
stop1, stop2, stop3: SIMPLE_STOP
```

Note the new syntax, enabling you to declare several entities of the same type together, rather than writing a declaration for each. You will just separate the entities by commas and write the type once after the colon.

The numbers correspond to the order on our line:



*Three stops on a
line*

This permits the next version of *build_a_line*:

```

build_a_line is
  -- Build an imaginary line and highlight it on the map.
  do
    Paris.display
    Metro.highlight
    -- Create the stops and associate each to its station:

    create stop1
    stop1.set_station (Station_Montrouge)

    create stop2
    stop2.set_station (Station_Issy)

    create stop3
    stop3.set_station (Station_Balard)

    -- Link each applicable stop to the next:
    stop1.link (stop2)
    stop2.link (stop3)
  
```

```

-- "Create fancy_line and give it the stops just created"
fancy_line.illuminate
end

```

Note how the pseudocode shrinks progressively as we add more instructions — “non-pseudo” code — that realize its intent. At the end we’ll have removed all pseudocode.

The two highlighted calls to *link* chain the first stop to the second and the second to the third. The third stop is not chained to anything; its *next* reference, set to void on creation, will remain void. That’s what we want since it represents the last stop on the line.

The calls to *set_station* must satisfy the precondition of this feature, which requires their arguments to be non-void. This is indeed satisfied by *Station_montrouge* and the other predefined stations from the “magic” of *TOUR*.

← “*Touch of Methodology: Precondition Principle*”, page 67.

6.5 CREATION PROCEDURES

Procedure *build_a_line* uses the simplest form of creation:

```
create some_stop
```

[C1]

for *some_stop* of type *SIMPLE_STOP*. This does the job but deserves an improvement. As the last version of the procedure indicates, the typical scheme for creating a stop associated with a station *existing_station* is in fact

```

create some_stop
some_stop.set_station (existing_station)

```

which in addition to the creation instruction requires a feature call immediately afterwards, to associate the new object to a station. The object resulting from the first instruction is useless because, as noted, it makes no sense to have a “stop” object without an associated station. We would like to reflect this through an invariant

```
invariant
```

```
station_exists: station /= Void
```

but then the class becomes incorrect since every instance must satisfy the invariant on creation, which won’t be the case here.

← “*Touch of Methodology: Invariant Principle*”, page 71.

So we have two separate reasons suggesting that the two instructions above, the creation and the call to *set_station*, should be merged into one:

- A reason of convenience: with the class as it stands, any client needing to create a stop must use both instructions; forgetting the second one will result in incorrect software and run-time failures. It's a general rule of software design that we should avoid producing elements that require specific prescriptions for use — “*When you do A, never forget to do B as well!*” — as it's all too easy for client programmers to miss the instructions. Better provide an operation that does everything needed, removing the need to learn a complicated interface.
- A reason of correctness: ensuring that instances of the class, straight from their creation, are consistent — here by specifying a non-void station.

To address both concerns, we may declare the class with one or more **creation procedures**. A creation procedure is a command that clients *must* call whenever they create an instance of the class, ensuring that the instance is properly initialized and, in particular, satisfies the invariant.

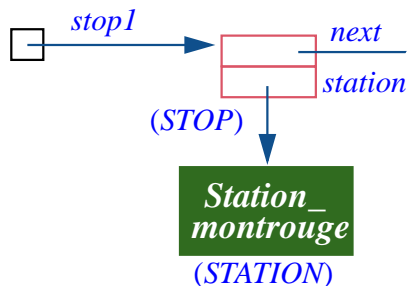
With a creation procedure, here *set_station*, and the stops now declared as

```
new_stop1, new_stop2, new_stop3: STOP
```

(rather than *SIMPLE_STOP* as before), the creation instruction as executed by clients is not just **create stop1** as before [1] but

```
create new_stop1 .set_station (Station_montrouge) [C2]
```

which has the effect achieved earlier by two separate instructions:



*After creation
instruction
using a creation
procedure*

The only difference between the two classes is that *STOP* has the desired invariant *station* \neq *Void* and declares *set_station* as a creation procedure. Here is how the class interface will look; other than the class names, only the highlighted parts have changed.:

```

class STOP create
    set_station
feature
    station: METRO_STATION
        -- Station which this stop represents
    next: STOP
        -- Next stop on same line
    set_station (s: METRO_STATION)
        -- Associate this stop with s.
    require
        station_exists: s /= Void
    ensure
        station_set: station = s
    link (s: STOP)
        -- Make s the next stop on the line.
    ensure
        next_set: next = s

invariant
    station_exists: station /= Void
end

```

Same as before,
now serves also as
creation procedure

At the top of the class interface we have a new clause

```

create
    set_station

```

using again the keyword **create**, and listing one of the commands of the class, *set_station*. This tells the client programmer that the class has one creation procedure, *set_station*. The clause lists one creation procedure; it could also list none, or several (since there may be more than one way to initialize a newly created object).

The consequence of including such a clause in the interface of the class is that it's no longer valid for a client to create an object using the basic form of the creation instruction, **create new_stop []**; because the class specifies creation procedures, you *must* use one of them, through form [2].

This rule enables the author of a class to force proper initialization of all instances that clients will create. It is closely connected with the notion of invariant: the requirement in this example is that every object will satisfy, immediately after creation, the desired invariant

```
station_exists: station /= Void
```

which is in turn ensured by the precondition of *set_station*. This is a general principle:

Touch of Methodology: Creation principle

If a class has a non-trivial invariant, it must list one or more creation procedures, whose purpose is to ensure that every instance, upon execution of a creation instruction, will satisfy the invariant.

“Non-trivial invariant” means any invariant other than *True* (which is usually omitted) or any property that would be ensured by letting all the fields take the default values ensured by the initialization rules (zero for numbers, *False* for booleans, *Void* for references).

Even in the absence of a strong invariant it may be useful to provide creation procedures to enable clients to combine creation with initialization. For example a class *POINT* describing points in a two-dimensional space may provide creation procedures *make_cartesian* and *make_polar*, each with two arguments denoting coordinates, enabling clients to create points identified by their cartesian or polar coordinates.

In some cases — such as this one — you may want to allow both forms, [1] and [2]. The technique then is to use

```
class POINT create
  default_create, make_cartesian, make_polar
feature
  ...
end
```

where *default_create* is the name of a feature (inherited by all classes from a common parent) with no arguments, which by default does nothing. To use this procedure you would normally write

```
create your_point.default_create
```

but this can be abbreviated into form [1], here

```
create your_point
```

which the **create** clause makes valid along with the other two forms

```
create your_point.make_cartesian (x, y)  
create your_point.make_polar (r, t)
```

6.6 OBJECT CREATION: SUMMARY

As a consequence of the preceding discussion, it's easy to remember what you must do to create an object:

Creating an instance of a class

- If the class has no **create** clause, use the basic form, **create x [1]**.
- If the class has a **create** clause listing one or more creation procedures, use **create x.make (...)**, where *make* is one of the creation procedures, and “(...)” stands for appropriate arguments for *make*, if any: the right number of arguments, with the right types, and guaranteed to satisfy the precondition if there's one.

6.7 CORRECTNESS OF A CREATION INSTRUCTION

For every instruction that we study, we must know precisely, in line with the principles of Design by Contract sketched in earlier chapters:

- How to use the instruction correctly: its *precondition*.
- What we are getting in return: the *postcondition*.

Together, these properties (complemented, for some constructs, by a notion of *invariant*) define the **correctness** of a language mechanism.

Here is the rule for the creation mechanism:

Touch of Methodology:
Creation Instruction Correctness Rule

The property (precondition) that must hold *before* a creation instruction:

1 • It is correct to execute the instruction if and only if the precondition of its creation procedure, if any, holds.

Properties (postconditions) that will hold *after* a creation instruction with target x of type C :

2 • $x \neq \text{Void}$ will hold.

3 • The postcondition of the creation procedure, if any, will hold.

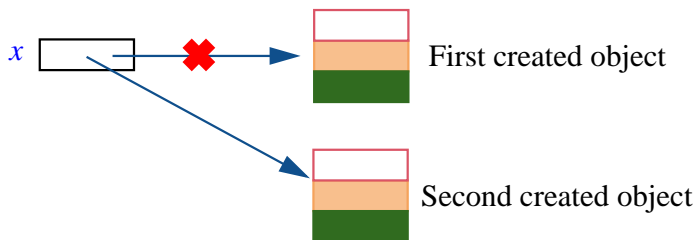
4 • The object attached to x will satisfy the invariant of C .

The form without creation procedure, **create x** , trivially satisfies clauses 1 and 3 since there is no applicable precondition or postcondition.

The correctness condition (clause 1) doesn't require x to be void. It is indeed not a mistake to create two objects successively for the same target x :

```
create x
  -- Here as a result x is not void (see clause 2)
create x
```

even though this form by itself is wasteful since the object created by the first instruction will be immediately forgotten:



Creating two objects in a row

The second creation instruction reattaches the reference x to the second object, so that the first object is now useless. (We'll see below what happens to such "orphan" objects.)

Although two successive creation instructions of the exact form shown make no sense, variants of this scheme can be useful. For example there could be other instructions between the two **create** *x*, doing something interesting with the first object. Or if a creation procedure is involved, as in **create** *x.make* (...), it may record the first object somewhere.

Clauses 2 to 3 define the effect of executing a creation instruction:

- Whether or not *x* was void before the creation instruction, it won't be void afterwards (clause 2) since the instruction attaches it to an object.
- If there is a creation procedure, its postcondition will hold for the newly created object (clause 3).
- In addition, that object will satisfy the **class invariant** (clause 4). Already stated in the **Invariant Principle**, this is an essential requirement on any creation instruction: to make sure that any object, when it starts out in life, satisfies the consistency condition that its class imposes on all instances, as expressed by the invariant. If the default initializations don't achieve this, then it is the duty of creation procedures to correct the situation by producing an initial state that satisfies the class invariant.

← "[Touch of Methodology: Invariant Principle](#)", page 71.

6.8 SYSTEM EXECUTION

A final consequence of the creation mechanism is that we can now find out what the process is for *executing* a system (an entire program).

Starting it all

With object creation, execution is in fact a simple concept:

Definitions:

System execution, root object, root class, root creation procedure

Executing a system consists of creating an instance — the **root object** — of a designated class from the system, called its **root class**, using a designated creation procedure of that class, called its **root creation procedure**.

The reason this suffices is that the root creation procedure (also called **root procedure** for short) may perform any actions that you have specified; in general it will itself create new objects and call other features, which may in turn do the same and so on. So you may think of your system — a collection of classes — as a set of balls on a billard table; the creation procedures kicks the first ball, which will hit other balls that in turn will kick more.

What's special about our billard boards (our systems) is that a ball, when kicked, can create new balls to be kicked, and that we may end up in a single execution with millions of balls rather than a dozen or so.

The root class, the system and the design process

The root class and root procedure are there to start a process that relies on mechanisms found in the classes of the system and their features. It's important to think of these classes as having a value of their own, independently of any particular system and its choice of root class and root procedure. As we have repeatedly seen, the classes are machines, each with its own role. A system is a particular assembly of such machines, where we have decided to choose one of them to start execution. But the classes exist beyond that system; a class may, for example appear in several systems, combined in each case with different classes.

A class that appears in many different systems because it provides features of general interest is said to be **reusable**; classes designed for reusability will be grouped into **libraries**. Even when designing specific applications rather than libraries you should always strive to make your classes as reusable as possible, since there's always the potential that you'll run into a similar need again.

→ Chapter 27 is devoted to reuse.

In older views of software engineering, a program was conceived as a monolithic construction consisting of a "main program" divided into "subprograms". This made it difficult to reuse some of the elements for new purposes, since they had all been produced as part of the fulfillment of one specific overall goal; it also hampered efforts to change the program if that particular goal changed, as it very often does in practice.

More modern approaches to software construction, based on the *object-oriented* ideas that we use in this book, fight these deficiencies by dividing the software into classes (a more general concept than subprogram) and encouraging the designer to give proper attention to each individual class, making it as complete and useful as possible. To obtain an actual system that handles a certain computer application you must, of course, select and combine a number of classes, and devise a root class and root procedure to kick off the execution process. In the end, then, the root procedure plays the traditional role of the main program. The difference is a methodological one: unlike a main program, the root class and root procedure are not a fundamental element of the system design; they are just a particular way to start off a particular execution process based on a set of classes that you've decided to combine in a particular way. But the prime focus is on these classes.

These observations point to some of the key concerns of professional software engineering (as opposed to amateur programming): **extendibility**, the ease with which it will be possible to adapt a system when user needs change over time; and **reusability**, the ease of reusing existing software for the needs of new applications.

Specifying the root

After this short foray into design principles let's come back to a more mundane issue: how will you specify the root class and root creation procedure of a system?

The development environment — EiffelStudio — is there to let you define such properties of a system. They are just part of the “Project Settings” of a system, which you can access through the [File → Project Settings](#) menu. A [section](#) of the EiffelStudio appendix gives the details.

→ [“SPECIFYING A ROOT CLASS AND CREATION PROCEDURE”, A.6, page 576](#)

The current object and general relativity

The perspective we have now gained on system execution enables us to understand a fundamental property of the object-oriented form of computation, which it might be tempting to call *general relativity* if the phrase hadn't already been taken, a while ago, by an ETH graduate . The question is a very basic one: when you see a name in a class, for example the attribute name *station* in class *SIMPLE_STOP*, what does it really mean? OK, in principle we know, if only through the declaration and its header comment:

```
station: METRO_STATION
-- Station which this stop represents
```

But what stop is “[this stop](#)”? In an instruction using the attribute, such as *station.set_name ("Louvre")*, of which station are we changing the name?

The answer can only be relative. The attribute refers to the *current object* at applicable times during execution. We can define this notion as follows:

Definition: Current object

At any time during the execution of a system, there is a current object determined as follows:

- 1 • The root object is, at the start of execution, the first current object.
- 2 • At the start of a qualified call $x.f(\dots)$, where x denotes an object, that object becomes the new current object.
- 3 • When such a call terminates, the previous current object becomes current again.
- 4 • No other operation causes a change of current object.

To denote the current object, you may use the reserved word *Current*.

So if you follow the execution of a system: the root object gets created; after possibly some other operations, in particular to create objects, it may perform a call using as its target one of these objects, which becomes current; it may again perform a call on another target, which will become current; and so on. Whenever a call terminates the previous current object resumes its role.

*Scheme for
system
execution*

This gives the answer to the question of what a feature name means: it denotes *the feature applied to the current object*. In class *SIMPLE_STOP*, any use of *station* — for example, *Console.show (station.name)* to display the name of a stop’s station — denotes the “station” field of the current *SIMPLE_STOP* object; this also explains “*this*” in header comments, as in “*Station which this stop represents*”.

This convention is central to the object-oriented style of programming. A class describes the properties and behavior of a certain category of objects. It does so by describing the properties and behavior of a typical representative of the category: the current object.

These observations lead us to generalize the notion of **call**. We know that an instruction or expression with a period, such as

<i>Console.show (station.name)</i>	-- An instruction
<i>station.name</i>	-- An expression

is a feature call, applied, like all calls, to a target object: the object denoted by *Console* in the first example and *station* in the second. But what about the status of *Console* and *station* themselves? They are calls too, with a target that is the current object. In fact you might also write them as

<i>Current.Console</i>
<i>Current.station</i>

where, as noted above, *Current* denotes the current object. You don't need, however, to use this *qualified* form in such cases; the *unqualified* forms *Console* and *station* have the same meaning. The definitions are as follows:

Definitions: Qualified and unqualified call

A feature call is **qualified** if it explicitly lists the target object, for example with dot notation, as in *x.f (args)*.

A call is **unqualified** if it doesn't list its target, which is then taken to be the current object, as in *f (args)*.

It is important to realize here that many expressions of whose status you may not have been quite sure until now are actually **calls** — unqualified. Examples as diverse (in the discussions so far) as uses of

<i>Paris, Louvre, Line8</i>	-- In our original class <i>PREVIEW</i> (chapter 2)
<i>sw_end, nw_end, i_th</i>	-- In the invariant of <i>METRO_LINE</i> (chapter 4)
<i>fancy_line</i>	-- In the present chapter

belong to this category. When the invariant of *METRO_LINE* stated

<i>sw_end = i_th (1)</i>

it meant that the Southwest end of the current metro line is the same as the first station of that same line.

← “Class invariants”,
page 70.

These observations show how fundamental and ubiquitous *calls* are in our programs. Along with qualified calls in dot notation, which clearly stand out as calls, simple innocuous-looking notations like *Console* or *Paris* are calls too, unqualified.

Calls are actually present in even more deceptive disguises: we'll see that an arithmetic expression like $a + b$ is, formally, just special syntax — in programming language jargon, “*syntactic sugar*” — for a qualified call $a.plus(b)$. →

Note that in the rule defining the current object above, case 4 tells us that operations other than qualified calls and returns don't change the current object. This is true of unqualified calls: where $x.f(args)$ makes the object attached to x the new current object for the duration of the call, the unqualified form $f(args)$ doesn't cause a change of current object. This is consistent with the above observation that you may also write it $Current.f(args)$. ← Page 131.

The “general relativity” nature of object-oriented programming can make you a bit dizzy at first (maybe it did until this section), since it prevents you from understanding program elements entirely by themselves: you must interpret them in terms of the enclosing class. It is a result of the modularity of the approach: its rejection of monolithic, all-in-one program architectures in favor of highly decentralized systems made of components to be developed autonomously and combined in many different ways.

6.9 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A reference is either *attached* to an object, or *void*.
- A feature call on an entity, such as $x.f(...)$, will only execute properly if the value of x is attached to an object.
- Every reference is initially void, and remains void in the absence of any operation such as creation that explicitly attaches it to an object.
- Void references serve to indicate missing information, and to terminate linked structures.
- A creation instruction of target x creates a new object and attaches x to it.
- The form of the creation instruction is **create** x , or — using a creation procedure p specified in the class — **create** $x.p(args)$.
- Prior to the execution of a creation procedure if any, the fields of a newly created object are initialized to standard default values, including zero for numbers and void for references.
- A creation instruction must ensure the invariant of the corresponding class. If the default initializations don't achieve this, the instruction must use a creation procedure that corrects the problem.
- Executing a system consists of creating an instance of a specified “root” class, with an associated root creation procedure.

- At any time of execution there is a *current object*: the object on which the routine last started operates.
- Calls can be qualified, applied to a target named explicitly, or unqualified, applied to the current object.
- Every unqualified mention of a feature must be understood (“general relativity” principle of O-O programming) as applying to an implicit object — the current object, a typical representative of the class.

New vocabulary

Attached	Creation procedure	Current object
Entity	Exception	Extendibility
Failure	Library	Main program
Qualified call	Reference	Reusable Reusability
Root class	Root creation procedure	(= Root procedure)
Root object	Unqualified call	Void reference

6-E EXERCISES

6-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

Talking about void

It may be useful to determine whether a reference is void or not. If and only if it is not a void reference, that is to say, x is attached to some object. A feature call $x.f(\dots)$ is permitted in the second case only. So when you need a condition such as

```
fancy_line.count >= 6
```

(stating, as an example, that a line has at least six stations), and you do not know for sure that *fancy_line* is attached, you should instead express it as

```
(fancy_line /= Void) and then (fancy_line.count >= 6)
```

guaranteeing that the expression will always evaluate properly at run time. For a non-existing line, it will yield *False*.

7

Control structures

We by now have a first grasp of the *data* structure of our programs, made of objects connected by references. It's time to look at the *control* structure, which determines the order in which the programs' execution will apply instructions to these objects.

7.1 PROBLEM-SOLVING STRUCTURES

You may have heard this satire of the reasoning skills supposedly taught to engineers:

How to boil a pot of water

- 1 • If the water is cold: put the pot on the fire, and wait until it boils.
- 2 • If the water is hot: wait until it cools down. Then — as the appropriate condition is now met — apply case 1.

As a water-boiling technique it may not be the most efficient, but it provides an example of combining some of the fundamental control structures:

- The *conditional*: “if this condition holds then do this, else do that”.
- The *sequence*: “do this and then do that”.
- The *routine*, which enables us to name a previously identified problem-solving technique (possibly parameterized), and reuse it in any applicable context.

Remembering the discussion of contracts in earlier chapters, you will also have noted that the throwback to case 1 in case 2 is only possible, as explicitly mentioned, because the first step of case 2 guarantees the **precondition** of case 1 (water is cold). Preconditions and other contract techniques will indeed play a large role in getting our control structures right.

In its light-hearted way, this example sets the proper context for our study of control structures: they are **problem-solving techniques**. To program is to solve a problem; each kind of control structure reflects a particular *strategy* for finding a solution to a problem.

The problem will always be expressed as: starting from known properties K , reach a certain goal G . In the example K is the property that we have a pot of water, and G that the water in the pot is boiling. The “strategies” provided by control structures are ways of reducing the problem to *easier* problems of that kind. For example:

- You may apply the **sequence** control structure if you find an intermediate goal I such that both of the following new problems are easier than the original (achieving G directly from K): achieve I from K ; achieve G from I . The sequence control structure applies, in order, a solution to the first new problem and a solution to the second one.
- The **conditional** control structure is the strategy of partitioning the set of possible initial situations, K , into two disjoint domains, so that it’s easier to solve the problem separately on each of these domains.
- The **loop** structure, of which we have yet to see an example, is the strategy of solving the problem on a subset (possibly trivial) of its domain and extending the solution repeatedly until it covers the whole domain.
- The **routine** control structure is the strategy of solving a problem by recognizing that it matches another problem — often of a more general nature — to which you already know a solution.

The **recursion** technique, important enough to occupy a chapter of its own, is applicable if you demonstrate that you can derive a solution by *assuming* a solution to the *same* problem applied to one or more smaller data structures.

→ Chapter 16.

Since programming is about solving problems, it will be particularly useful to study these and other control structures in this light.

For each of the control structures we will successively explore:

- The general idea, through *examples*.
- The *syntax* of the corresponding language construct.
- Its *semantics*: the run-time effect, in this case how the control structure governs the order of execution of the instructions it contains.
- *Correctness* rules needed to ensure that the semantics is what we want — that executing the control structure produces a meaningful result rather than a program crash or some other unpleasant consequence.

7.2 THE NOTION OF ALGORITHM

Control structures take care of scheduling the operations in the processes carried out by computers. Such processes are called *algorithms*; this is one of the fundamental concepts of computing science. You may have seen the term already, even in the popular press which nowadays discusses things like “*cryptographic algorithms*” in reporting security issues. For the study of control structures we need a precise understanding of the concept.

Example and definition

In general terms an algorithm is a *description* of a computational process, sufficient to enable a machine — for our interests, a computer — to carry out the process on any input data without further instructions.

You already know many algorithms. To add two integers, as in

$$\begin{array}{r} 687 \\ + 42 \\ \hline = 729 \end{array}$$

you apply the following rules (probably without thinking of them explicitly):

Touch of Elementary Maths: Adding two decimal numbers

The process consists of a number of *steps*, each working on a **position** in the numbers. The position for the first step is the position of the rightmost digit of both numbers; for each subsequent step, it's the position immediately to the left of the previous one.

At each step, there is a **carry**. The initial carry is 0.

At each step, let m be the digit from the first number at the step's position and n the corresponding digit from the second number, with the convention that if either number has no digit at that position the corresponding value (m or n) is 0.

At each step, the process performs the following:

- 1 • Compute the sum s of m , n and the carry.
- 2 • If s is less than 10, write s at the step's position in the result line, and let the carry for the next step be 0.

- 3 • If s is 10 or more, write $s - 10$ (which is a single digit, as s cannot be more than 19) at the step's position in the result line, and let the carry for the next step be 1.

The process stops when there are no digits at the step's position on either line and the carry is 0.

Operation 3 relies on an assumption: “ s cannot be more than 19”. Without it the process would not make sense, since we want to write digits only. To guarantee the correctness of the algorithm we have to prove that the property holds at every step. Indeed, m and n are at most 9 each, so their sum is at most 18; at the first step the carry is 0, and at every following step it can only (as a result of that same operation 3) be either 0 or 1, so its sum with m and n will at most be 19. This is an example of an **invariant property**, a concept that we'll study in more detail with loops.

Precision and explicitness: algorithms vs recipes

Although less precise than the standard for publishing algorithms, the preceding specification is more punctilious than most of the prescriptions we are used to follow — with, it must be said, varying degrees of success — in ordinary life. Here is for example a trilingual set of directions on a bag of pre-cooked minestrone:

PREPARAZIONE E TEMPI DI COTTURA
ZUBEREITUNG - PREPARATION

Versate le verdure ancora surgelate in 1 litro abbondante d'acqua fredda con 2 cucchiari d'olio, salate e cuocete secondo i tempi indicati.

Tiefgefrorene Gemüse in einen Liter kaltes Wasser geben, 2 Esslöffel Öl und Salz hinzufügen.

Verser les légumes surgelés dans 1 litre d'eau froide, ajouter deux cuillers à soupe d'huile et du sel.



*Not an algorithm
 (see English
 translation in
 text)*

Source: Buitoni

In German and French the instructions state: “*Pour the frozen vegetables into one liter of cold water, add two tablespoonfuls of oil and salt*”. What’s striking is not so much the lack of precision (“tablespoonful” can be given an exact conventional value, and anyway the idea of using such a general term is that it doesn’t matter too much whether you take a slightly bigger or smaller tablespoon) as the absence of the key instruction: if you want to get an edible result, you’d better heat up the thing at some point. Only the Italian version mentions this detail — “*cook according to the times given*” — without which the pictures would be meaningless.

For such instructions intended for human interpretation, lack of explicitness is not an issue; it will be immediately clear to most readers that they can’t prepare such food without heating it, and that the pictures indicate cooking times. (Even I succeeded!) But such an approach would not work for an algorithm. You must specify every operation, every detail of the process; and you must specify them in a form that leaves no room for ambiguity.

Properties of an algorithm

For algorithms, as opposed to informal recipes, we expect a number of properties captured by the following definition.

Definition: Algorithm

An algorithm is the specification of a process acting on a (possibly empty) set of data, satisfying the following five rules:

- 1 • The specification defines the applicable sets of data.
- 2 • The specification defines a set of elementary actions, from which all steps of the process are drawn.
- 3 • The specification defines the possible order or orders in which the process may carry out these steps.
- 4 • The specification of the elementary actions (rule 2) and of the permitted orderings (rule 3) relies on precisely defined conventions, allowing the process to be carried out by an automaton (such as a computer) without human intervention, with results for the same set of data guaranteed to be the same on two different automata following the same conventions.
- 5 • For any set of data to which the process is applicable (as per rule 1), the process is guaranteed to terminate after executing a finite number of the algorithm’s steps.

The above method for adding two numbers possesses the required properties:

- 1 • It describes a process to be applied to some data, and specifies the kind of data: two integers.
- 2 • The process relies on well-defined basic actions: set a value to zero or to a known number, add three numbers, compare a number to 10.
- 3 • The description specifies in what order to apply such actions.
- 4 • It is precise. That precision should be sufficient for any two people to understand and apply the algorithm in the same way, although, as noted, possibly not sufficient for other goals.
- 5 • For any applicable data — two numbers written in decimal notation — the process will terminate after a finite number of steps. This is intuitively clear but must be ascertained rigorously; we'll see how to do this by showing that the quantity $M - \text{step} + 1$ is a *variant*.

→ "[Loop termination](#)", page 159.

Algorithms vs programs

You may wonder, in light of the preceding definition, what distinguishes an algorithm from a program. The basic concept is indeed the same.

It is sometimes said that the difference is the *abstraction* level: that a program is meant to execute on a particular machine, whereas an algorithm is an abstract definition of a computing process, independent of any computing devices. This made sense a few decades ago, when programs were expressed in low-level codes for specific computers. Algorithms then served to express the *essence* of programs: the computing process described independently from any computer. But that view is no longer applicable today:

- To express *programs*, we can use clear, high-level notations, defined at a level of abstraction far above the details of any particular computer. The Eiffel notation used in this book is an example.
- To express an *algorithm* in a way that fully meets the definition's requirements, in particular the requirement of precision — condition 4 — we will need a notation with rigorously defined syntax and semantics, making it in the end equivalent to a programming language.

It is true that practical descriptions of algorithms often leave unspecified some details, such as choices of data structures, which a program cannot omit since it wouldn't then compile and execute. This practice does seem to suggest that algorithms are more abstract than programs. But it is only a useful convention to facilitate publication; for that purpose, it renounces some of the precision that true algorithms require (condition 4 again), and every reader of the description understands that to get an algorithm in the official sense she would need to bring the missing details back in.

So we can't rely on the level of abstraction to distinguish algorithms from programs. Two differences — or nuances — are perhaps more significant:

- An algorithm describes a single computing process. A program in the traditional sense was also that — “*Compute this month's payroll!*” — but programs today involve *lots* of algorithms. We've already seen several in the Traffic system (display a line, animate a line, display a route...) and there are hundreds more. It's to emphasize this variety of algorithms that when talking about such a combination of software elements I tend to use, rather than “program” (which may still suggest the idea of doing one possibly one task), the word *system*.
- As important to a program as the description of the processing steps is the description of the data structure — in the object-oriented approach of this book, the **object structure** — to which they apply. This criterion is not absolute either, since you can't really separate the algorithmic steps from the structure which they manipulate. But in describing programming concepts we may sometimes want to emphasize the processing aspect — the algorithm in a narrow sense of the term — and sometimes the data. This explains the title of a famous programming book by Niklaus Wirth: *Algorithms + Data Structures = Programs*.



Wirth

(Picture to be replaced.)

Prentice Hall, 1976.

The object-oriented approach to software construction has the peculiarity of giving a central role to the data through the object types: the classes. *Every* algorithm is then attached to a particular class. Eiffel applies this rule without exception: every algorithm that you write will appear as a *feature* of some class. This approach is justified by considerations of software quality that we'll explore in part **IV**. It implies, for this book, that we will study the algorithm and data aspects in close connection.

Control structures, as studied in this chapter, are one example of an algorithmic concept not directly related to a particular kind of data structure.

7.3 CONTROL STRUCTURE BASICS

The specification of an algorithm must include elements of two kinds:

- The elementary steps to execute (clause 2 of the definition of “algorithm”). ← Page 141.
- The order of their execution (clause 3).

Control structures handle the second of these needs. Precisely:

Definitions: Control flow, control structure

The scheduling of a program’s operations during execution is called its **control flow**.

A **control structure** is a program construct affecting the control flow.

Or “flow of control”.

There are, as previewed, three fundamental forms of control structure:

- The **sequence**, consisting of instructions listed in a certain order; its execution consists of executing these instructions in the same order.
That’s the control structure we have been using implicitly in all the examples so far, since we have been writing instructions under the assumption that they would be executed in the order given.
- The **loop**, containing a sequence of instructions to be executed repeatedly.
- The **conditional**, consisting of a condition and two sequences of instructions; its execution consists of executing one or the other of these sequences depending on whether the condition — a boolean expression — evaluates to **True** or **False**. It can be generalized to a choice between more than two possibilities.

These are mechanisms for scheduling the execution of our program’s instructions, taking advantage of three fundamental capabilities of computers:

- Executing *all* of a set of specified actions, in a specified order.
- Executing a *single* specified action, or some variants of it, *many times*.
- Executing *one* of a set of specified actions, depending on a specified condition.

Such control structures assume that the program will at run time be doing at most one thing at a time. With several computers, or with a single computer sharing its time between different programs, we can actually have *parallel* execution, yielding more control structures. These are the topic of a later chapter, devoted to *concurrency* (“concurrent” means the same as “parallel”).

→ Chapter 23.

Our basic control structures can be combined without restriction, so that you may for example have a conditional involving two sequences of instructions, where one or more of the instructions in these sequences are in turn loops, or conditionals, themselves involving further sub-structures.

Such a description of a computing process, consisting of instructions grouped into control structures describing their run-time scheduling, constitutes an algorithm.

When you have defined an algorithm, you will often want to wrap it into a program unit with a name, which you can then use through that name. Such a grouping is known as a **routine**, a fundamental form of program structuring, achieving on the control side what classes give us on the data side. Routines enable you in effect to add *new* control structures to the available repertoire, by abstracting particular combinations of existing structures.

→ *Routines are studied in the next chapter.*

These notions are the subject of the following sections. In addition we will review two other forms of control structuring:

- The *branching instruction*, also known as **goto** (“Go to” written as one word), which has fallen from grace as a tool for programmers — we’ll see why — but still plays a role in computer instruction codes.
- **Table-driven control**, a way to specify control through the *data* structure rather than through explicit control instructions.

→ [“THE LOWER LEVEL: BRANCHING INSTRUCTIONS”, 7.7, page 180.](#)
→ [“TABLE-DRIVEN CONTROL”, 7.9, page 194.](#)

7.4 SEQUENCE (COMPOUND INSTRUCTION)

We are all familiar with solving a problem by identifying one or more intermediate goals, so that we can proceed in steps. If there is only one such intermediate goal we’ll solve two separate problems:

- Achieve the intermediate goal from the hypothesis.
- Achieve the problem’s overall goal from the intermediate goal.

More generally, with n intermediate goals we’ll have $n + 1$ steps, where step i (for $2 \leq i \leq n$) has to achieve the i -th intermediate goal from the preceding one.

Examples

In our application domain of city travel, a typical example of sequence is a possible strategy going from a place a to a place b :

- 1 • On the map, find the metro station ma closest to a .
- 2 • On the map, find the metro station mb closest to b .
- 3 • Walk from a to ma .
- 4 • Take the metro from ma to mb .
- 5 • Walk from mb to b .

This is a human strategy, not a program. A program might, for example, build a route from *a* to *b*. Assuming the declarations

```
walking_1, walking_2, metro_1: SEGMENT
full: ROUTE
```

you might use the sequence of instructions

```
-- Version 1
create walking_1.make_walk (a, ma)
create walking_2.make_walk (mb, b)
create metro_1.make_metro (ma, mb)
create full.make_empty
full.append (walking_1)
full.append (metro_1)
full.append (walking_2)
```

This takes advantage of the following creation procedures of class *SEGMENT*:

- *make_walk*, producing a walking segment from one place to another.
- *make_metro*, producing a metro segment from one place to another.

and the following features of class *ROUTE*:

- The creation procedure *make_empty*, producing an empty route.
- The command *append*, adding a segment at the end of a route.

Programming time! **Creating and animating a route**

Using the above scheme, write and execute a system that will create a route from *Elysee_palace* to *Eiffel_tower* (both place names defined as features in class *TOURISM*), and animate the route.

Put the corresponding software elements, and the remaining ones for this chapter, in a new class called *ROUTES*.

For this and future programming exercises, you won't any more get a step-by-step description of how to write, compile and run the example, unless this involves some EiffelStudio mechanism that you haven't seen yet. All the necessary techniques have been seen before; if you have any hesitation consult the EiffelStudio [appendix](#) or go back to the earlier examples.

→ [A, page 573](#).

Compound: syntax

As the above example shows, the sequence control structure is not new with this chapter: we have seen it many times before — in fact, ever since our very first program example — without having a name for it. We simply wrote several instructions in the intended order of execution, as in ← Feature *explore*, page 20.

```
Paris.display
Louvre.spotlight
Metro.highlight
Route1.animate
```

Since it is often useful to consider such a sequence of instructions as a single instruction — for example to make it part of a bigger control structure — it’s also called a **compound instruction**, or just “compound”.

The syntax rule is very simple:

Syntax: Compound instruction

To specify a sequence, or *compound*, of zero or more instructions, write them one after the other in the desired order of execution, optionally separated by semicolons.

We haven’t used semicolons so far. The style rule indeed suggests not to bother with them:

Touch of Style: Semicolons between instructions

- If (as should almost always be the case) successive instructions appear on separate lines, **omit** the semicolon.
- In the occasional case of two instructions appearing on the same line (to be used only for very short instructions and if there’s a good reason to save on the number of lines), **always** separate them by a semicolon.

So if you will be printing out the above “**Version 1**” example and are down to your last roll of paper (or have a very environmentally-conscious boss), you might write the last three instructions as

```
full.append(walking_1) ; full.append(metro_1) ; full.append(walking_2)
```

but there is seldom a reason to do so. Instead, you will usually have one line per instruction; then you can just forget the semicolons.

It is important to remember that the separation into lines does **not** by itself carry any semantic value; line return is just a “break” character, which has the same effect as a space or a tab. So nothing prevents you from writing

← From “*Breaks and indentation*”, page 49.

```
full.append(walking_1)full.append(metro_1)full.append(walking_2)
```

Ugly!

Nothing, that is, except good taste, elementary common sense, the official style rules, and any hint of a trace of a shadow of a tinge of concern for whoever is going to try to read your program later, including two readers of particular interest: the instructor (if you’re taking a course); and — after a few days, weeks or months — yourself.

Even on separate lines, some people are initially nervous about omitting the semicolons, perhaps because many commonly used programming languages have strict rules requiring them in many places and prohibiting them in others, causing the program texts to indulge in an orgy of semicolons, and the compiler to harass you if you forget one, or put one where it’s not expected. To get over semicolon addiction a simple test suffices: put two versions of the same program side by side, both with a single instruction per line but one with semicolons and the other without; you’ll see right away that the second one is cleaner and more readable.

If you do use semicolons, mistakenly including an extra semicolon will be harmless, because *instruction_1 ; ; instruction_2* is formally understood as *three* instructions, of which the second is an *empty* instruction. So this won’t cause any trouble or error. All the same, it’s better to clean up your code and remove any unneeded element.

Compound: semantics

The run-time behavior of a sequence is what the name of this control structure and the earlier informal discussion suggest:

Semantics: Compound instruction

Executing a sequence of instructions consists of executing each instruction in turn, in the order given.

Note that the syntax description talks of “zero or more” instructions (not one or more) and hence permits empty sequences; the semantics in this case is to do nothing. This is not a very exciting case, but it is useful to allow it for when a sequence appears as part of a larger structure.

Order overspecification

You may have noticed that in the above example (“[\[1\]](#)”) the chosen sequence is only one of a number of possibilities. For example we could append each segment to the full route as soon as we’ve created it: ← *Page 146.*

```

-- Create the route:                                     -- Version 2
create full.make_empty

-- Create and append the first segment:
create walking_1.make_walk (a, ma)
full.append (walking_1)

-- Create and append the second segment:
create metro_1.make_metro (ma, mb)
full.append (metro_1)]

-- Create and append the third segment:
create walking_2.make_walk (mb, b)
full.append (walking_2)

```

Many other orders are possible; the only constraints for this example are that any instruction using an object (route or segment) must come after the creation instruction for that object, and that we append segments in the proper order.

Using the sequence control structure often creates such cases of **overspecification**, that is to say, of a solution that is not the most general possible one. This kind of overspecification does not directly harm the software, but one must be conscious that the solution is only one of a set of possibilities. In some circumstances, when execution speed is a concern, it may be possible to speed up execution by executing some group of instructions *concurrently* with others, that is to say, in parallel. For example in “Version 1” the four initial creation instructions can be executed concurrently without affecting the result. Concurrency, however, is a delicate matter and programmers usually don’t explicitly prescribe it for such elementary cases.

→ See chapter [23](#) about concurrency.

Compound: correctness

We have seen that a feature may have a contract including a precondition and a postcondition. These properties govern *calls* to the feature, such as the above call *full.append (walking_1)*. The precondition tells the client (the calling feature) what it *must* guarantee to be correctly serviced. The postcondition tells the client what it *may* assume on termination of such a correct call.

Since the instructions making up a Compound will be executed in the order given, there is an obvious correctness rule for this control structure:

Correctness: Compound instruction

For a Compound instruction to be correct:

- The program must ensure that the precondition of the Compound's first instruction, if any, holds prior to any execution.
- The postcondition of every instruction in the compound must imply the precondition of the following one if any.
- The postcondition of the last instruction must imply the postcondition desired for the entire Compound.

Special case: an empty Compound is (by itself) always correct, but achieves no new postcondition.

In our example, you may check the contract for the feature *append* of class *ROUTE* by bringing up the class in EiffelStudio. With some postcondition clauses omitted, it reads

```
append (s: SEGMENT)
  require
    segment_exists: s /= Void
  ensure
    lengths_added: count = old count + s.count
```

Every creation instruction of the form **create** *x* or **create** *x.make (...)* ensures that the condition *x /= Void* will hold after its execution. So our example satisfies the correctness rule for compound instructions; this is true in both its “[1]” and its “[2]”. But that wouldn't be the case if we changed the order of the first instructions to start with

← Clause 2 of “[Touch of Methodology: Creation Instruction Correctness Rule](#)”, page 127.


```

-- Version 3 (erroneous)

-- Create the route:
create full.make_empty

-- Create and append the first segment:
full.append (walking_1)
create walking_1.make_walk (a, ma)

```

where *walking_1*, at the place of use, denotes a non-existent *SEGMENT* object that the extract mistakenly attempts to append to the route.

7.5 LOOPS

Our second control structure, the loop, taps into one of the most amazing features of computers: their ability to repeat an operation, or variants of that operation, many times — *very* many times by human standards.

A typical example of loop is an animation scheme to highlight a metro line by displaying a red dot on each of its stations in turn, for half a second. The system *Show_line* in the Traffic delivery does this. You can execute it now if you wish; the effect at one of the intermediate steps is this:

Highlighting a station

Here is a loop that achieves this effect. It uses *show_spot* (*p*) to display a red spot at point *p* on the screen for a few seconds. To understand the details we'll need concepts introduced in the rest of this discussion, so you should just take this example as an introduction to how a loop looks:

```

from
    Line8.start
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
end

```

The loop moves a “cursor” (a virtual marker) to the beginning of the line (*start*), then until the cursor is beyond the last position (*is_after*) it performs the following for each successive station (*item*): display the red spot at the *location* of the station, and advance the cursor to the next station through command *forth*. The value of *Spot_time* is predefined to half a second.

This shows some of the key ingredients of a loop: initialization (**from**), exit condition (**until**), and actions to be repeatedly executed (**loop**). To get a full understanding of this loop we must first study the concepts in more depth.

→ In [“Animating a metro line”](#), page 164.

Programming time! **Animating Line 8**

Put the preceding loop in a feature *traverse* of class *ROUTES* (the example class for this chapter). For this example and subsequent variations, update the class and run the system to observe the results.

Loops as approximations

As a problem-solving technique, the loop is the method of approximating the result on successive, ever bigger subsets of the problem space.

In the Metro Line Animation example, the problem is to display a red dot on each station of the line. Successive approximations are: display a dot on no station at all; display it on the first station; display it successively on the first two stations; and so on.

Here is another example. Assume you want to know the maximum of a set of one or more values N_1, N_2, \dots, N_n . The following strategy, described informally, will work:

- Define *max* to be N_1 . It is then true, trivially, that *max* is the maximum of the set of values containing just one value, N_1 .
- Then for every successive $i = 2, \dots, n$ do the following: if N_i is greater than the current *max*, redefine *max* to be N_i .

This ensures that at the i -th step (where the first step is the first case, the second step is the second case for $i = 2$, the third step for $i = 3$ etc.) the following property, called a **loop invariant** of the loop, holds:

Loop invariant of the “maximum” strategy, at step i

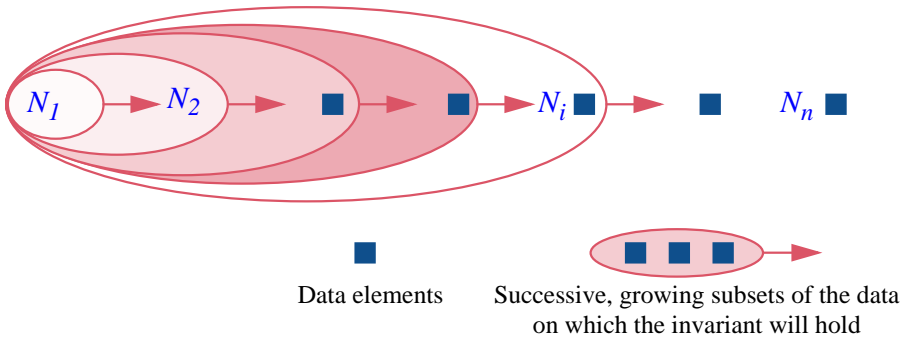
max is the maximum of N_1, N_2, \dots, N_i

At the n -th step, where $i = n$, this gives us that

“ max is the maximum of N_1, N_2, \dots, N_n ”

which is the desired result.

The following picture illustrates the loop strategy in this case:



Finding a maximum by successive approximations

The loop establishes the invariant property “ max is the maximum of the first i values” for a trivial value: $i = 1$; then it repeatedly extends the subset of the data on which the invariant holds.

In the Metro Line Animation example, the invariant would be: “A red dot has been displayed on all the stations visited so far”.

The notion of invariant is not new, since we have already encountered class invariants. The two forms of invariant are related, since both describe a property which certain operations must maintain (meaning: if executed in a state satisfying the invariant, they must terminate in a state satisfying it again). But their roles are different: a class invariant applies to an entire class and must be maintained by the execution of *features* of the class; a loop invariant applies to a single loop and must be maintained by each iteration of the *loop body*.

← “Class invariants”, page 70.

The loop strategy

Although many loops are more sophisticated, the “maximum” example illustrates the general form of loops as a problem-solving strategy.

The strategy is useful when a problem consists, starting from some initial property *Pre*, of establishing a certain goal *Post* characterizing some set of data *DS*. This set is finite, although it might be very large.

To use a loop is to find a weaker (more general) form of the goal *Post*: a property *INV*(*s*) — the loop’s invariant — defined on subsets *s* of *DS* (not just *DS* itself), with the following properties:

- L1 • You know an initial subset *Init* of *DS* such that the initial condition *Pre* implies *INV*(*Init*); in other words, the invariant holds for the initial subset.
- L2 • *INV*(*DS*), that is to say *INV* applied to the whole set, implies your goal *Post*.
- L3 • You know a technique, applicable when *INV*(*s*) holds for a set *s* that is not yet all of *DS*, to make *INV*(*s'*) hold for a larger subset *s'* of *DS*.

The “maximum” example has all these ingredients: *DS* is the set of numbers $\{N_1, N_2, \dots, N_n\}$; the precondition *Pre* is the property that *DS* has at least one element; the goal *Post* is the property that we have found the maximum of these numbers; and the invariant *INV*(*s*), where *s* is a subset N_1, N_2, \dots, N_i of *DS*, is that we have found the maximum of *s*. Then:

- M1 • If *Pre* is satisfied, that is to say there’s at least one number, then we know an initial subset *Init* such that *INV*(*Init*) holds: just take the set consisting of only the first number N_1 .
- M2 • *INV*(*DS*) — the invariant applied to the whole set $\{N_1, N_2, \dots, N_n\}$ — does imply the goal *Post*; actually, it’s identical.
- M3 • When *INV*(*s*) holds for a set $s = \{N_1, N_2, \dots, N_i\}$ which is not all of *DS* — in other words, $i < n$ — then we can establish *INV*(*s'*) for a larger subset *s'* of *DS*: we just take *s'* to be $\{N_1, N_2, \dots, N_i, N_{i+1}\}$, and the new maximum to be the greater of the previous maximum and N_{i+1} .

Note — in the general case — how carefully the invariant is devised to fit our general strategy of solving a problem by successive approximation:

- *INV* is sufficiently **weak** that we can establish it easily for some initial subset, usually very small, of the whole data set.
- It is sufficiently **strong** to give us the entire desired goal, *Post*, when applied to the whole set.
- It is sufficiently **flexible** to let us extend it from any applicable subset to a slightly larger one.

By repeatedly performing this extension, having started by establishing the invariant on the initial subset, we’ll get to the desired result. This strategy of successive approximations of the goal on progressively larger sets might take many iterations, but computers are fast, and they don’t go on strike to complain of repetitive work.

These observations define how the loop works as a control structure. Its execution will:

- S1 • Establish *INV* (*Init*), taking advantage of L1. This gives us *Init* as a first subset *s* on which *INV* holds.
- S2 • As long as *s* is not the complete set *DS*, apply the technique of L3 to establish *INV* on a new, larger *s*.
- S3 • As soon as *s* is the whole of *DS*, stop: we have established *INV* on *DS*, which thanks to L2 establishes our goal *Post*.

This process is guaranteed to terminate because we always assume *DS* to be a finite set; since *s* is always a subset of *DS*, and grows by at least one element at every step, it has to reach the full *DS* after a finite number of iterations of step S2. In some cases, however, establishing termination will require some extra care.

← Page [154](#).

→ “[Loop termination](#)”, page [159](#).

Loop instruction: basic syntax

To express the loop strategy as a program text we will use, in the “maximum” example, the following general structure:

```

from
    -- “Define max to be  $N_1$ ”
    -- “Define i to be 1”
until
    i = n
loop
    -- “Redefine max as the greater of the current maximum and  $N_{i+1}$ ”
    -- “Increase i by one”
end

```

Here all the basic instructions are still pseudocode rather than actual Eiffel. The example illustrates the three required parts of a loop construct (complemented below by two *optional* parts):

- The **from** clause introduces the initialization instructions (S1).
- The **loop** clause introduces the instructions to be executed in each of the successive iterations (S2).
- The **until** clause introduces the exit condition: the condition under which the iterations will terminate (S3).

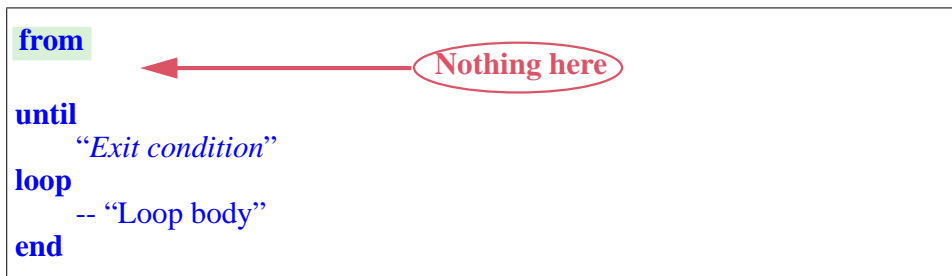
The run-time effect of this construct, suggested by the keywords (**from**, **until**, **loop**), is in line with the previous discussion:

- First, execute the instructions in the **from** clause (*initialization*).
- Then execute the instructions (*body*) in the **loop** clause until the condition stated in the **until** clause (*exit condition*) holds.

The last point means more precisely that after initialization the body will be executed:

- Not at all, in the case the exit condition holds immediately after the initialization.
- Once, if one execution of the body leads to the exit condition being true.
- More generally: i times for some i , if the exit condition will be false after j executions of the body for $1 \leq j < i$, and true after i executions.

Syntactically, the **from** and **loop** clauses each contain a compound instruction. As a consequence you may include any number of instructions, including zero. It indeed happens that a loop doesn't require an explicit initialization instruction (in cases when the context before the loop already implies the invariant); then the **from** clause will be empty:



This doesn't apply to the **loop** clause, since it must make some progress in the approximation (bring at least one new element to the subset s of the previous discussion); otherwise the loop process wouldn't terminate. So you can't have an empty **loop** clause in a realistic loop.

Including the invariant

The basic form of loop, as just seen, doesn't show the loop invariant. This is unpleasant since the invariant is essential to understanding what the loop is about. The optional but recommended **invariant** clause takes care of this. With this clause our example becomes:

```

from
  -- "Define max to be  $N_1$ "
  -- "Define i to be  $1$ "
invariant
  -- "max is the maximum of  $\{N_1, N_2, \dots, N_i\}$ "
until
   $i = n$ 
loop
  -- "Redefine max as the greater of the current maximum and  $N_{i+1}$ "
  -- "Increase i by one"
end

```

The invariant in this example is still pseudocode, but useful nonetheless to convey essential information about the loop.

Loop instruction: correctness

The invariant of a loop has two characteristic properties:

Correctness: **Loop invariant**

The invariant of a loop must be:

- L1 • Ensured by the initialization (**from** clause)
- L2 • Preserved by the body (**loop** clause) whenever executed with the exit condition not satisfied.

An instruction “preserves” a property if its execution, started with that property satisfied, terminates with the property satisfied again. It's this preservation property that explains the name “invariant”.

The *class* invariant must similarly be ensured upon instance creation, and preserved by every feature of the class.

← “[Class invariants](#)”,
[page 70](#).

As we have seen, the purpose of a loop is to achieve a certain outcome by successive approximations. The steps towards this goal are the initialization and then successive executions of the body. After each one of these steps, a property holds that is an approximation of the final desired outcome; that's the invariant. In our two examples the invariants, in pseudocode, are:

- “*max* is the maximum of $\{N_1, N_2, \dots, N_i\}$ ” as the i -th approximation, for $1 \leq i \leq n$, of the final property “*max* is the maximum of $\{N_1, N_2, \dots, N_n\}$ ”.
- “A red spot has been displayed on all stations visited so far, in their order on the line”, as an approximation of the final property that the spot has been displayed on all stations in order.

When the loop execution terminates, the invariant *Loop_invariant* will still hold because of properties **L1** and **L2** above. In addition, of course, the exit condition *Loop_exit* will hold — otherwise the loop wouldn't have terminated yet. So the final condition produced by the loop is

Loop_invariant **and** *Loop_exit*

This is the outcome achieved by the loop:

Correctness:
Loop Postcondition Rule

The condition achieved by the execution of a loop is the conjunction of its invariant and its exit condition.

The syntax highlights the Loop Postcondition Rule by putting the **invariant** and **until** clauses next to each other. So if you see a loop with these two elements and want to know what it does, just look at these two adjacent clauses:

```

from
  -- “Define max to be  $N_1$ ”
  -- “Define i to be  $I$ ”
invariant
  “max is the maximum of  $\{N_1, N_2, \dots, N_i\}$ ”
until
   $i = n$ 
loop
  -- “Redefine max as the greater of the current maximum and  $N_{i+1}$ ”
  -- “Increase i by one”
end

```

and Final condition

The effect of the loop is their conjunction (their **and**).

In the Metro Line animation example, the exit condition is, informally, “**all stations have been visited**”; conjoined with the invariant stated above, this tells us that a red dot has been displayed on all stations, in the order of the line.

The Loop Postcondition Rule is of course the direct consequence of how loops were defined in the first place, as an approximation mechanism. Quoting from that earlier discussion, the idea was to choose as invariant a generalization of the final goal, which is: ← Page [154](#).

- “*Sufficiently **weak** that we can establish it easily for some initial subset of the whole data set*”: this is the role of the initialization.
- “*Sufficiently **flexible** to let us extend it from any applicable subset to a slightly larger one*”: this is the role of the body, executed when the invariant is satisfied and the exit condition is not satisfied; it then yields a state where the invariant is satisfied again.
- “*Sufficiently **strong** to give us the entire desired goal when applied to the whole set*”: this is achieved on exit, as per the Loop Postcondition Rule, by the conjunction of the exit condition and the invariant.

Loop termination

The loop execution scheme, as described, repeatedly performs the loop body until the exit condition is satisfied. If the loop derives from a well-devised approximation strategy as above, its execution will terminate after a finite number of iterations, since the set being approximated is finite and each iteration adds a new element to its approximation, so that the process cannot go forever. But the loop syntax permits an arbitrary initialization, exit condition and loop body, so it could in principle execute forever, like ← As noted on page [155](#).

```

from
  “Any instruction here (or none at all)”
until
  0 /= 0
loop
  “Any instruction here (or none at all)”
end

```

In this extreme example the exit condition can never be satisfied, so the loop can't ever terminate. If you execute the corresponding program, you'll be sitting at the terminal with nothing happening; after a while you'll probably realize that something's wrong, so you'll interrupt the program (EiffelStudio has a button for that purpose). But of course this is very disturbing, especially since you have no way to know — if you are just a user of the program, and have no access to its text — whether the program is really looping forever, or just taking a long time.

A loop with $0 \neq 0$ as exit condition makes no sense, but even when you mean well you may inadvertently produce a non-terminating loop and hence a non-terminating program. To avoid this unpleasant result the best technique is to ensure that each loop you write has an associated **loop variant**:

Definition: Loop variant

A variant for a loop is an integer expression possessing the following properties:

- 1 • On execution of the loop initialization (**from** clause), the variant has a non-negative value.
- 2 • Every execution of the loop body (**loop** clause) when the exit condition is not satisfied decreases the value of the variant.
- 3 • Every such execution also keeps the variant non-negative.

If indeed you can find such an expression, then you have shown that the loop will terminate after a finite number of iterations: it's not possible for a non-negative integer value to decrease forever while remaining non-negative. In fact, if we know the original value V of the variant after initialization, we know that the loop will terminate after at most V iterations, since each iteration decreases the variant by at least 1.

For this reasoning to hold, the variant must indeed be an integer. Real numbers would not work, since it is perfectly possible (in mathematics, if not on a computer) for an infinite sequence of real numbers, such as the sequence $1, 1/2, 1/3, \dots, 1/n, \dots$, to consist of ever decreasing values.

If you know a variant, the syntax lets you specify it in a **variant** clause next to the **invariant** clause. For example we may add a specification of the loop variant to our computation of the maximum:

```

from
  -- “Define max to be  $N_1$ ”
  -- “Define i to be 1”
invariant
   $1 \leq i$ 
   $i \leq n$ 
  -- “max is the maximum of  $\{N_1, N_2, \dots, N_i\}$ ”
variant
   $n - i$ 
until
   $i = n$ 
loop
  -- “Redefine max as the greater of the current maximum and  $N_{i+1}$ ”
  -- “Increase i by one”
end

```

The variant identified here is $n - i$. It indeed satisfies the conditions:

- V1 • The initialization sets i to 1. The program assumes that $n \geq 1$. So the variant is initially non-negative.
- V2 • The loop body increases i by one, and hence decreases $n - i$ by one. Because the invariant combined with the negation of the exit condition tells us that i remains less than n at the start of the loop body, the variant remains non-negative.
- V3 • When the exit condition is not satisfied, i will be less than n and hence $n - i$, when decreased by one, will remain non-negative.

For the last point **V3**, it is not sufficient to consider the negation of the exit condition, which only tells us that $i \neq n$: we need to be sure that $i < n$. But note the new invariant properties added above: $1 \leq i$ and $i \leq n$. These are ensured by the initialization and preserved by the body when executed with $i \neq n$, so they are indeed invariant. Then when the exit condition is not satisfied, that is to say, $i \neq n$, we know from the invariant property $i \leq n$ that in fact $i < n$.

You may well feel at this point that I am splitting hairs and that the loop as given is evidently correct — that it will always terminate, having computed the maximum of the given set of values. But in practice it’s a common mistake to write a loop that will not terminate. If you have ever used a program to see it “hang”, it might very well be the result of such a mistake on the part of its author. Maybe the problem didn’t appear in the program tests; tests can only capture a small part of all possible cases. Only through the kind of reasoning illustrated above can you guarantee — for your own programs — that a loop will *always* terminate, regardless of the program inputs.

Considering the possibility of non-termination leads to important notions which you will study in more detail in a course on the theory of computation:

Touch of Theory:
The Halting Problem and undecidability

The prospect of a loop that runs forever is disturbing. Isn't there a way, given a program, to check that every loop in it will terminate? Compilers already perform some other verifications for us, in particular *type checks* (if x is of type *METRO_STATION* and you write a feature call $x.f$, the compiler will issue an error message and refuse to compile your program unless f is a feature of class *METRO_STATION*). Perhaps they could also check loop termination?

The answer to this general question is *no*. A theorem states that — assuming a programming language powerful enough for practical needs — it's *impossible* to write a program (such as a compiler) that will correctly report, when fed any program text, whether that program will always terminate. This is known as the **undecidability** of the **Halting Problem**:

- The *Halting Problem* is whether a program will terminate (halt).
- A problem is *undecidable* if no effective technique exists that will yield a correct solution in every case.

The Halting Problem is the most famous undecidability result in the theory of computation, although not the only one.

Depressing as it may sound, this result doesn't prevent you in practice, when you write a program, from guaranteeing — as you should! — that it will terminate. The undecidability theorem rules out any *general* mechanism that would determine termination for *any* program, but not *specific* techniques for demonstrating that *some* programs will terminate. The use of an explicit loop variant is such a practical technique — a very effective one. If you can prove that an integer expression has the variant properties **V1** to **V3** (initially non-negative, decreased, and maintained non-negative by every iteration), then you have the guarantee that the loop will terminate.

Commercial-grade compilers are not yet able to perform such proofs, so you will have to do them manually by inspecting the program, and, if there's any doubt, let EiffelStudio check at run time that the variant decreases on each iteration. Unlike the general Halting Problem this is not a fundamental impossibility, but a limitation of current technology.

We will actually be able to *prove* the undecidability of the Halting Problem once we've studied routines.

→ "[AN APPLICATION: PROVING THE UNDECIDABILITY OF THE HALTING PROBLEM](#)", page

Touch of History: **Tackling the Halting Problem**

The Halting Problem was described — as a special case of the “decision problem”, or *Entscheidungsproblem*, a general issue going back to Leibniz in the 17th-18th century and Hilbert in the early 20th — and its undecidability proved, a decade before the appearance of actual stored-program computers, in a famous mathematical paper of 1936, “*On Computable Numbers, with an Application to the Entscheidungsproblem*”.

The author, the British mathematician Alan Turing, relied on an abstract model for a computing machine, known as the **Turing machine**. The Turing machine — a mathematical concept, not a real machine — is still used today to discuss general properties of computation, independent of any particular computer architecture or programming language.

Turing didn't stop at mathematical machines. He went on during the second World War to lead the successful effort to decrypt the German cryptographic machine, the Enigma, and afterwards to build several of the world's first actual computers. (The end of his life was marred by — let's resort to understatement — lack of recognition of his achievements by the authorities of his country.)

Alan Turing introduced many of the seminal ideas of computing science. The highest distinction in the field, the Turing Award, honors his memory.

The undecidability of the Halting Problem belongs to a series of *negative* results that burst into one science after another in the early 1900s, spoiling the Great Science Party that the new century had seemed to herald:

- Mathematicians saw the validity of set theory — and, as it turned out, of the basic techniques of logical reasoning — put into question by the emergence of apparent *paradoxes*; just as an enormous 30-year effort to repair the foundations, by such mathematicians as Bertrand Russell and David Hilbert, seemed to have a chance of succeeding, Kurt Gödel proved that in any axiomatic system powerful enough to describe ordinary mathematics there will be properties that can be neither proved nor disproved. This **incompleteness** theorem is one of the most striking examples of the limitations on our ability to *reason*.
- At about the same time, physics had to accept the Heisenberg uncertainty principle and other results of quantum mechanics that put limits on our ability to *observe*.

Undecidability results, for the Halting Problem in particular, are the computing science version of such absolute limitations.

The theoretical undecidability of the Halting Problem shouldn't directly affect — except for the emotional trauma of coming to terms with our intellectual limitations, but I trust you'll recover — your practice of programming; yet non-terminating programs are not just a theoretical possibility but a very real threat. To avoid its unpleasant occurrence, the advice is clear:

Touch of Methodology: Loop termination

Whenever you write a loop, examine the question of its termination. Convince yourself — by identifying a suitable variant — that it will always have a finite number of iterations. If you can't, rework the loop until you can equip it with a satisfactory variant.

Animating a metro line

As a simple example of loop, we come back to the problem sketched at the beginning of this section: “animating” Line 8 by having a red dot move through its stations. We may use: ← Page [151](#).

- From class *METRO_STATION*, a query *location*, indicating the station's place on the map; the result is of type *POINT*, representing the notion of point in a 2-dimensional space.
- A command *show_spot* from class *TOURISM*; *show_spot (p)*, for *p* of type *POINT*, will display a red spot at location *p*.
- *Spot_time*, also from *TOURISM*, a predefined value for the time to leave the red spot on each station; it's set to 0.5 seconds.

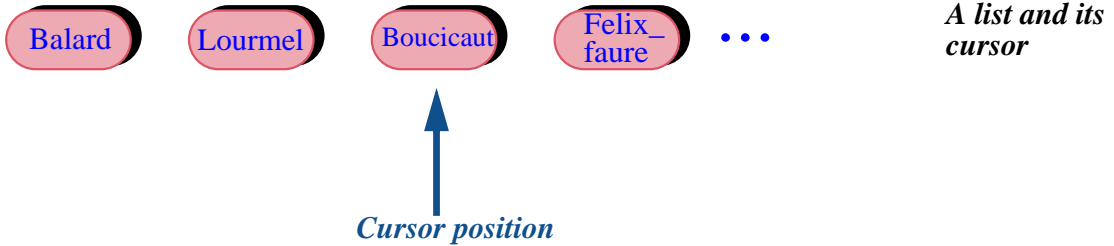
The task of the loop will be to call *show_spot* at the position of every station of the line, in sequence.

To get to the successive stations we could (with the help of operations on integer variables studied in the chapter after next) use the query *i_th* which gives us the *i*-th element of a line, through the call *some_line.i_th (i)*, for any applicable *i*; the loop would have to perform → Assignment:
chapter [9](#).

```
show_spot (Line8.i_th (i).position)
```

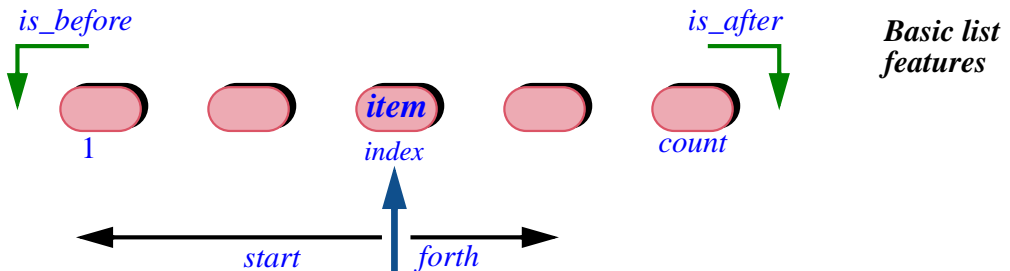
for successive values of *i*, ranging from 1 to *Line8.count*. Instead we'll use this opportunity to discover a typical form of loop used for *iterating* over object structures such as list. To “iterate” over a data structure is to perform an operation on each of its elements, or on a subset of its elements selected by an explicit criterion. Here the operation consists of calling *show_spot* for the position of the selected station.

Classes such as *METRO_LINE*, and in general classes describing ordered lists of things, support iteration by letting you move a **cursor** (a marker) to successive places in the list. The cursor doesn't have to be an actual object, simply an abstract notion denoting, at any point in time, a position in the list:



In the state shown, the cursor is on the third station. *METRO_LINE* and other list classes include the following four key features — two commands and two queries — for iterating over the corresponding object structures:

- The command *start*, which brings the cursor to the first item. (An item is an element of the list, such as a station in the case of a Metro Line).
- The command *forth*, which advances the cursor to the next item.
- The query *item*, which yields the item, if any, at cursor position.
- The boolean query *is_after*, yielding *True* if and only if the cursor is at the extreme right, past the last element if any. For symmetry there's also *is_before*, although we don't need it yet.



Also useful is the query *index* which, as illustrated, gives the index of the current cursor position, which is 1 for the first item and *count* for the last.

This is enough to give us the general iteration scheme for lists, and its application to our example:

```

from
    Line8.start
invariant
    -- “For all stations before cursor position, a spot has been displayed”
    -- “More invariant clauses (see below)”
variant
     $Line8.count - Line8.index + 1$ 
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
end

```

The scheme using *start*, *forth*, *item* and *is_after* to iterate over a list occurs so frequently that you must make sure to understand its details and convince yourself of its correctness. Informally, its effect is clear:

- Bring the cursor to the first item of the list, if any, through the call to *start* in the initialization.
- At each step through the loop, display for *Spot_time* seconds a red spot on the station *Line8.item* at cursor position.
- Also at each step, after displaying the spot, advance the cursor by one position, through *forth*.
- Stop when the cursor *is_after*, that is to say, past the last item.

To avoid any confusion (I hope the previous discussion doesn't leave room for any, but just in case...): be sure to note that there is no connection between the position of a station on the map or, as we've called it, its **location**, and the notion of **cursor position**:

- A station has a geographical location in the city, determined by its coordinates.
- The cursor exists only in our imagination and in our program, not in the world out there. It's an internal marker enabling the program to iterate over a list of stations, remembering from one iteration to the next what item it visited last.

Programming time! Terminating and non-terminating loop

Update the loop in feature *traverse* of class *ROUTES* to read as the last version, with the variant and (informal) invariant. Run it.

Now remove the instruction *Line8.forth*, introducing an error. Run the system again and observe what happens.

(Then restore the missing line for future exercises.)

Let's now consider the loop constituents in more detail. The initialization uses *start* to bring the cursor to the first position. In the Contract View of class *METRO_LINE* (and of any similar class based on the notion of list) you may see that the specification of *start* reads:

```

start
  -- Bring cursor to first element
  -- (No effect if empty)
ensure
  at_first: (not is_empty) implies (index = 1)
  empty_convention: (is_empty) implies (is_after)

```

The boolean query *is_empty* indicates whether the list is empty. Let's consider first the case of a non-empty list (like *Line8*). The first postcondition clause *at_first* of *start* indicates that after initialization the cursor is on the first element (*index* = 1), as we would expect.

The loop's exit condition is *Line8.is_after* and so for a non-empty list it won't be satisfied after initialization; you can in fact check this through the clause in the class invariant that reads

```
is_after = (index = count + 1)
```

Since this is an equality between two boolean values, it means that *is_after* is true if and only if *index* = *count* + 1; for a non-empty list *count* will be at least 1, so after the initialization, when *index* = 1, it's impossible for *is_after* to hold. So in this case the loop body will be executed at least once.

Each execution of the loop body performs

```
show_spot ( Line8.item .location)
```

which displays a red spot at the *location* of the item at the current cursor position in the list.

Programming time! **Using the debugger**

As you read through the complementary explanations of this example, and in particular its correctness arguments, it is useful to get a concrete picture by following what's going on at run time. The EiffelStudio **debugger** provides this capability. Use it to execute the program as a whole, or instruction by instruction in the feature *traverse* of class *ROUTES*, and to stop it at any time, then traverse the object structure and examine the contents of relevant objects.

For example you can see the instance of *METRO_LINE* and check that the results of queries such as *is_before* and *is_after* agree with the expected values as deduced from the analysis of the program carried out below.

Such a run-time inspection tool is not a substitute for systematic reasoning about programs. Reasoning yields the properties that will hold in all executions of the program; run-time inspection can only tell you that a particular property holds at one point of one execution. But it's still very helpful as a way to gain a practical understanding of what's going on; it lets you, literally, *see* your program as it's executing.

As its name indicates, the debugger helps, when a program doesn't function as expected, to find out what the error — the *bug* — is. But its scope is broader; bug or no bug, it gives you a direct window into program execution. Don't wait until something goes wrong to take advantage of it.

A section of the EiffelStudio appendix tells you how to run the debugger to examine the execution of the current example.

→ Chapter 26 covers testing and debugging.

→ "CONTROLLING EXECUTION AND INSPECTING OBJECTS", A.7, page 576.

Understanding and verifying the loop

Let's gain a deeper understanding of our loop example by verifying that it's correct. It's a good idea, as you read, to use the debugger to examine the objects involved at various stages of the execution.

For ease of reference here is the loop again:

```

from [1]
    Line8.start
invariant
    -- “For all stations before cursor position, a spot has been displayed”
    -- “More invariant clauses (see below)”
variant
     $Line8.count - Line8.index + 1$ 
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
end

```

Let’s first deal with the case of an empty list. As noted, the postcondition of the command *start* reads

```

ensure
    at_first: (not is_empty) implies (index = 1)
    empty_convention: empty implies is_after

```

so that — by “convention” — an empty list will, after a call to *start*, satisfy *is_after*. Of course this convention is not there by accident: it’s precisely meant to ensure that the typical iteration on a list, using the form illustrated by our example — start with *start*, exit on *is_after*, and each time through the loop do something with *item* and then move on with *forth* — stops immediately, having produced no visible effect, when applied to an empty list. This is indeed the case for our loop.

Touch of Methodology: Beware of the border cases!

Extreme cases, such as an empty list, are a frequent source of errors. It’s all too easy, when you design your program, to think only of non-empty cases (and *test* it only on those). Then once in a while the execution of the program might use an empty structure, and fail. The problem doesn’t just arise for empty structures but for extreme cases in general; another example is a structure with limited capacity, which might cause problems when *full*.

When designing a program and reasoning about its correctness, make sure to think of the extreme cases, and to verify that your reasoning holds for these cases as well as the more ordinary ones. This also applies to testing: always include extreme cases in your program tests.

→ Chapter 26 discusses testing.

You may run the example on an empty line (class *TOURISM* defines a feature *Empty_line* for that purpose) and use the debugger to follow what happens. For the rest of this discussion we assume that the line is not empty.

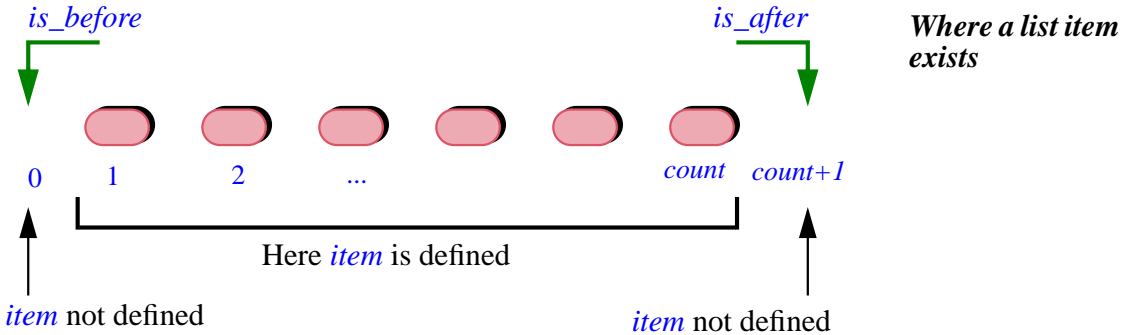
Bring up the specification for *item*. You'll see that it has a precondition, stating that the query is only applicable if the cursor is on a list element:

```

item: METRO_STATION
  -- Current item
  require
    not_off: not (is_after or is_before)

```

as suggested by the figure:



Since the loop body calls *item* — in the call to *show_spot* — we must verify that prior to the execution of this call the precondition will always hold.

Note first that the exit condition is **not** *is_after*, so *is_after* is not true when *show_spot* is called (if it were, the loop body would not be executed). Next, *is_before* must also not be true. This is ensured by adding the following property to the loop invariant:

```

not_before_unless_empty: (not is_empty) implies (not is_before) [I2]

```

Why is this indeed a loop invariant? We note in the class invariant that

```

is_before = (index = 0)
index >= 0
index <= count + 1

```

in other words, *is_before* is true, obviously enough, if and only if the cursor's *index* position is zero. After initialization, the postcondition of *start* — clause *at_first* as given above — indicates that *index* is one, so *is_before* cannot hold. ← Page 167. Then we have to check that every execution of the loop body preserves this property [2]. The specification of *forth* reads

```
forth
  -- Move cursor to next position
require
  not_after: not is_after
ensure
  moved_forth: index = old index + 1
```

Since *index* has been increased by one, it cannot be zero, and hence *is_before* cannot hold. So [2] is indeed a loop invariant.

You should track the properties just seen on an actual execution of the loop; use the debugger to execute the loop iterations one by one, and explore the object structure at each step.

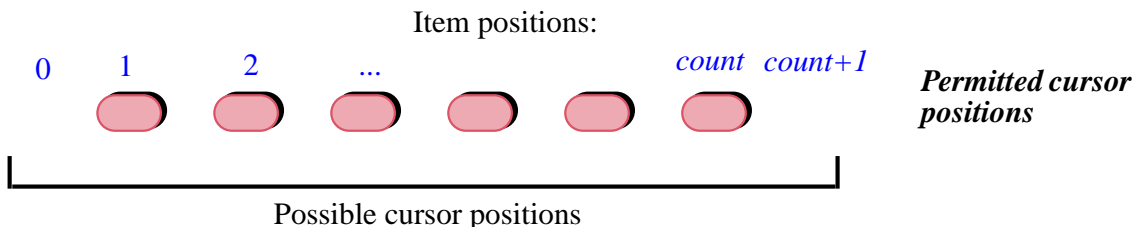
The cursor and where it will go

To complete our understanding of loops and of this example it's useful to check a little further into the class invariant of *METRO_LINE*. If you bring it up you'll see the following two clauses, also appearing in all the library classes having to do with list structures of any variation:

```
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

This expresses, as illustrated below, that we allow the cursor to be:

- On an item if any (if the list is empty there are no items)
- One position left of the first item, but no further to the left.
- One position right of the last, but no further right.



Being able to go off by one position is useful for the loop scheme illustrated by our spot-moving example:

```

from
    some_list.start
invariant
    -- “All items left of cursor, if any, have been processed”
variant
    some_list.count – some_list.index + 1
until
    some_list.is_after
loop
    -- “Process item at cursor position”
    some_list.forth
end

```

After the loop has processed the last item, the highlighted call to *forth* will move past that item. This will cause *is_after* to be true, so that there will be no further iteration; but it is essential that the call to *forth* be possible even though it leads to a position (at *count* + 1) where there is no list item. The invariant permits this; it is matched by the precondition of *forth*, cited above:

```

require
    not_after: not is_after

```

7.6 CONDITIONAL INSTRUCTIONS

The next control structure, the conditional instruction, doesn’t raise as many issues as the loop, but is also a fundamental building block for programs.

A conditional instruction involves a condition and (in the basic form) two instructions; it will execute one of these instructions if the condition holds, the other one if not.

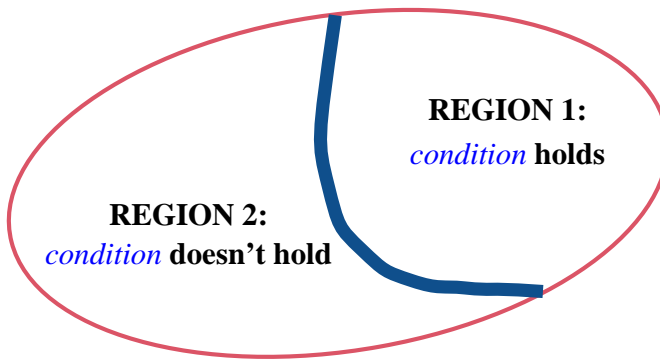
As a problem-solving technique, the conditional instruction corresponds to *separating cases*: divide the problem space into two (or more) parts such that it is easier to solve the problem separately in each part. For example, when trying to get from the Eiffel Tower to the Louvre:

- *If* the weather is good and you are not too tired, *then* walk to the nearest station and take the metro.
- *Else* try to catch a taxi.

Or, in elementary mathematics, if you are asked for real roots of the quadratic equation $ax^2 + bx + c = 0$:

- If the discriminant δ defined as $b^2 - 4ac$ is positive, then you can derive the two solutions $(-b \pm \sqrt{\delta}) / 2a$.
- Else, if δ is zero, then you can derive the single solution $-b / 2a$.
- Else, there is no real solution (only complex ones).

You may picture the use of a conditional, as a problem-solving technique, in terms of a partition of the set of cases to handle:



Conditional as a partition of the problem space

You have found a separation of the problem space into two parts, characterized by a certain *condition*, which holds on one and not in the other, such that it's easier to find a separate solution for each part than a global solution. The basic form of the construct will be:

```

if condition then
    "Produce Region 1 solution"
else
    "Produce Region 2 solution"
end

```

Conditional: an example

As a typical example of conditional instruction, let's adapt our last loop [example](#). The loop was displaying a spot on each station. We refine this by stopping a little longer, with a spot that blinks, on exchanges. Class *TOURISM* obligingly provides for that purpose a command *show_blinking_spot*, complementing *show_stop* used so far.

← [\[1\], page 169.](#)

We can achieve the result through this variation of the previous loop, where the only change is the highlighted part:

```

from [3]
    Line8.start
invariant
    not_before_unless_empty: (not is_empty) implies (not is_before)
    -- “For all stations before cursor position, a spot has been displayed”
variant
    Line8.count – Line8.index + 1
until
    Line8.is_after
loop
    if Line8.item.is_exchange then
        show_blinking_spot (Line8.item.location)
    else
        show_spot (Line8.item.location)
    end
    Line8.forth
end

```

The example conditional instruction uses three times the expression *Line8.item*, a query call. It is more elegant to compute the result once, give it a name, and then reuse that name whenever needed. We’ll learn soon how to do this.

→ Assignment:
chapter 9.

Programming time! **Using a conditional**

Update the preceding example — feature *traverse* in class *ROUTES* — to take into account the conditional instruction above. Run the result.

For the conditional instruction we need no less than four new keywords: **if**, **then** and **else**, as well as **elseif** which will appear next. The basic structure is straightforward:

```

if condition then
    Compound_1
else
    Compound_2
end

```

where *condition* is a boolean expression and *Compound_1* and *Compound_2* are compound instructions — sequences of zero or more instructions.

← “SEQUENCE (COMPOUND INSTRUCTION)”, 7.4, page 145.

Conditional structure and variants

Being sequences of *zero* or more instructions, both *Compound_1* and *Compound_2* may be empty, so that you may write

```

if condition then
    Compound_1
else
    ← Nothing here
end

```

Not the recommended style (see next).

with nothing in the **else** part. This corresponds to the frequent case of an instruction or sequence of instructions that you want to execute only if a certain condition holds, doing nothing otherwise. Rather than including an **else** clause with no instructions you may in this case omit the clause altogether. You will just write:

```

if condition then
    Compound_1
end
    ← No else clause

```

Recommended style.

In either form — with or without an **else** clause — any of the instructions making up the compounds can itself be a control structure, for example a loop or another conditional.

Assume for example that you want to do something different — yet — for a Metro station that connects to the railway network. You may use this scheme as a replacement for the previous loop:

```

from ... invariant ... variant ... until ... loop
    -- The omitted loop clauses are as in [3] above
if Line8.item.is_exchange then
    show_blinking_spot (Line8.item.location)
else
    if Line8.item.is_railway_connection then
        show_big_red_spot (Line8.item.location)
    else
        show_spot (Line8.item.location)
    end
end
Line8.forth
end

```

[4]

Not the recommended style; see [7], page 178

Such inclusion of program structures within others is called **nesting**. Here we have a conditional instruction nested in another conditional instruction, itself nested in a loop.

Touch of Style: How deep a nest?

There are no theoretical limits on how deeply you may nest control structures. The limits are practical: good taste, and the desire to keep your programs readable.

The last example [4] has a depth of *four*: basic instructions appearing within a control structure, itself within a structure, itself within another. This is about the maximum that you should use in ordinary programming. That's not an absolute rule: some algorithms genuinely require a higher depth of nesting. But when you reach such a level you should ask yourself whether you can avoid the extra nesting.

The alternative, in such a case, is usually to carve out a significant part of the structure and give it an independent status as a *routine*, replacing its original occurrence by a *call* to that routine. We'll study routines next.

→ Chapter 8.

In examples such as the last one [4] the depth of nesting makes the structure appear more complex than it needs to be, and we'll be able to simplify it without recourse to routines. This simplification is applicable to conditionals repeatedly nested in the **else** part of other conditionals:

```

if condition1 then                                     [5]
  ...
else
  if condition2 then
    ...
    else
      if condition3 then
        ...
        else
          ...
          -- "More nested occurrences of if ... then ... else ... end"
        end
      end
    end
  end
end

```

In this structure the nesting gives a deceptive impression of complexity, whereas in fact the decision structure is sequential:

- If *condition*₁ holds, execute the first **then** part, and nothing else.
- For $i > 1$, if *condition*_{*i*} holds but none of the *condition*_{*j*} for $j < i$, execute the *i*-th **then** part, and nothing else.
- If no *condition*_{*i*} holds, execute the innermost **else** part, and nothing else.

The keyword **elseif** enables you to remove the unnecessary nesting in this case by writing the successive cases at the same level:

```

if condition1 then                                     [6]
  ...
elseif condition2 then
  ...
elseif condition3 then
  ...
elseif ... More conditions if needed ... then
  ...
else           -- As before, the else part is optional
  ...
end

```

This replaces a *Matrioshka*-like structure



Matrioshki
(Russian dolls)

Conditional: syntax

Here is a summary of the form of conditional instructions:

Syntax: Conditional

A conditional instruction consists, in order, of:

- An “If part”, of the form **if** *condition*.
- A “Then part” of the form **then** *compound*.
- Zero or more “Else if parts”, each of the form **elseif** *condition* **then** *compound*.
- Zero or one “Else part” of the form **else** *compound*
- The keyword **end**.

Here each *condition* is a boolean expression, and each *compound* is a compound instruction.

If, by the way, you find this form of syntax description too verbose and at the same time not rigorous enough (for example we have to understand that each *condition* denotes a separate boolean expression), you are right. The better technique for describing such non-trivial syntax constructs — such as the control structures of this chapter — is a mathematical notation known as BNF. We’ll learn it in the chapter on syntax. The informal specifications of the present chapter, aided by examples, will suffice in the meantime.

→ Chapter 13.

Conditional: semantics

The effect of the conditional instruction reflects the preceding discussions:

Semantics: Conditional

The execution of a conditional instruction consists of executing at most one of the compound instructions appearing in its “Then part”, “Else if” parts if any and “Else part” if any, determined as follows:

- If the condition following **if** has value *True*, the compound in the Then part.
- If that condition has value *False*, the first compound in an Else if part, if any, such that the corresponding condition has value *True*.
- If none of the above applies and there is an Else part, its compound.
- If none of the above applies, no compound (the conditional has no effect).

Conditional: correctness

The correctness of a conditional instruction is the separate correctness of both of its branches under the respective assumption that the condition holds and doesn't hold:

Correctness: Conditional instruction

For a conditional instruction **if c then a else b end** to be correct, the program must ensure that prior to the conditional's execution:

- If c holds, the precondition of a holds.
- If c doesn't hold, the precondition of b holds.

The postconditions of a and b — each executed under these conditions — must imply the postcondition desired for the conditional instruction.

7.7 THE LOWER LEVEL: BRANCHING INSTRUCTIONS

The combination of our three fundamental mechanisms — sequence, loop and conditional — provides the appropriate basis (when complemented by routines) to build the control structures that we need to build our programs.

These programming-language mechanisms have counterparts in the instruction codes directly executed by computers, or “machine languages”. Compilers are responsible for the mapping between the two. But the control structures offered by most machine languages are more rudimentary than those we have studied.

Conditional and unconditional branching

Machine-level control mechanisms typically include:

- **Unconditional branch:** an instruction that transfers control to the instruction found at a given location in memory. In the example below this instruction will appear as **BR $Address$** where $Address$ is the location of the target instruction.
- **Conditional branch:** transfer control to a specified location if two specified values are equal, otherwise proceed to the next instruction. We may write it **BEQ $Value1$ $Value2$ $Address$** . The name stands for “**B**branch if **E**qual”.

With these mechanisms the equivalent of

```

if  $a = b$  then
    Compound_1
else
    Compound_2
end

```

looks like this:

```

100    BEQ loc_a loc_b 111
101    ... Code for Compound_2
...
110    BR 125
111    ... Code for Compound_1 ...
...
125    ... Code for continuation of program ...

```

A conditional in machine code

Here *loc_a* and *loc_b* stand for memory locations holding the values of *a* and *b*. The numbers on the left are instruction locations; starting at 100 is just an example, and so are the numbers of locations occupied by the code for each compound instruction. Determining precisely the space taken up by machine instructions associated with every program element, and laying out everything in memory, can be a tricky task; since almost no one writes application at the machine-language level, this task is the responsibility of compilers (that is to say of compiler writers) rather than application programmers.

*Such values on which machine instructions operate directly are usually held in special locations called **registers**. See chapter [12](#).*

From this *conditional* example, you can infer the code structure that a compiler would generate for a *loop*. This is the subject of an exercise.

→ [7-E.2, page 195](#).

The goto instruction and flowcharts

Branching instructions, conditional and non-conditional, reflect basic operations that computers are able to perform: test certain boolean conditions such as the equality of two values held in memory; transfer control to an instruction stored at a specified location. So it is natural that we should find these instructions in machine language. But they were not always confined there. All programming languages used to have, and many still offer, a **goto** instruction, whose name comes from “go to” written as a single word. In such languages you may give a *label* to instruction, as in

```

some_label: some_instruction

```

where `some_label` is a name — an identifier — of your choice. It is common for such languages to use a colon `:` between the label and the instruction it labels, but other conventions are possible. These labels correspond to the location numbers (`100`, `101`, ...) appearing in our machine-language example, but they are chosen by the programmer, who lets the compiler maps them to actual locations. The language then includes an instruction of the form

```
goto label
```

whose effect is to transfer control — which would otherwise continue with the instruction appearing next — to the instruction with the given `label`.

Instead of an `if condition then Compound_1 else Compound_2 end` conditional, the language may have a more limited choice instruction `test condition simple_instruction`, which executes the `simple_instruction` if the `condition` is true, otherwise proceeding sequentially. This closely reflects machine-level instructions such as `BEQ`. To express the equivalent of the conditional in such a language you would write:

The keyword is usually `if` in such languages; `test` is used here to avoid confusion with the full-blown conditional instruction.

```

test condition goto else_part
Compound_1
goto continue
else_part: Compound_2
continue: ... Continuation of program...
```

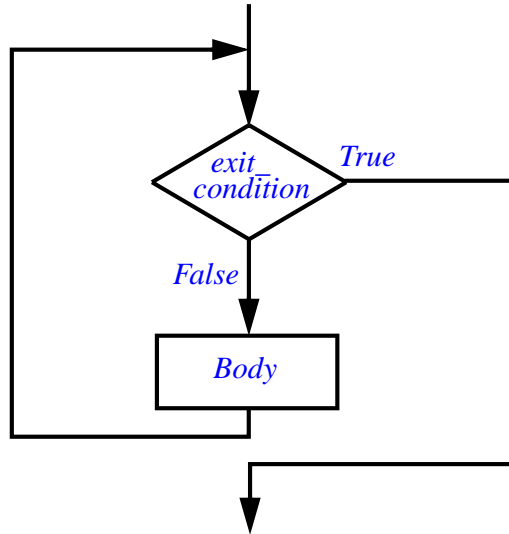
This is less clear than the conditional instruction, with its hierarchical, symmetric structure.

The comparison is even less favorable for a loop which, ignoring the `from` part, would be represented as:

```

start:      test exit_condition goto continue
            Body
            goto start
continue:   ... Continuation of program...
```


with its control flow involving two **goto** branches going in opposite directions:



Flowchart for a loop

Flowcharts

The last figure uses a representation of the control flow called a **program flowchart** or just flowchart. The box shapes are standardized: diamond for a test node, here with two outgoing branches for *True* and *False*; rectangle for a processing block, here for the *Body*. You may test your understanding of the concept by drawing a flowchart for the Conditional construct.

→ *Exercise 7-E.3, page 196.*

Flowcharts used to be a popular way of expressing the control structure of a program. Nowadays you may still encounter them in descriptions of non-software processes, but for programming they have fallen into disrepute (to the point that some authors call them “*flaw charts*”). That’s easy to understand. When programming languages gave you, as control structures, the unconditional **goto** and a conditional branching instruction such as **test condition goto label**, flowcharts provided a welcome high-level view of the run-time flow of control, clearer than what could be inferred from reading the program text with its succession of branching and non-branching instructions. But this is obsolete for two reasons:

- Our programs do more complicated things. We nest compounds within loops within conditionals; big flowcharts quickly become messy.
- The mechanisms of this chapter — compound, loop, conditional — provide a higher form of expression for the control structure. A neatly formatted program text, with indentation clearly reflecting the nesting, carries a better representation of the run-time scheduling of instructions.

The move from flowcharts to carefully chosen and properly nested control structures belies the cliché that “a picture is worth a thousand words”. In software we need many thousands or indeed millions of “words”, but it’s critical that they be the *right* words. For precise, unambiguous descriptions pictures lose their appeal.

The correctness of a program may depend on fine details such as using a condition $i \leq n$ rather than $i < n$; the best pictures in the world are largely helpless when it comes to getting such aspects right.

Goto harmful?

Flowcharts are not the only casualty of the reexamination of control structure specification that occurred as software engineering grew into a discipline: the **goto** instruction also lost favor.

The reasons are pretty much the same. The mechanisms that we have studied offer better control over execution. This comment actually contains two separate arguments which we must distinguish carefully:

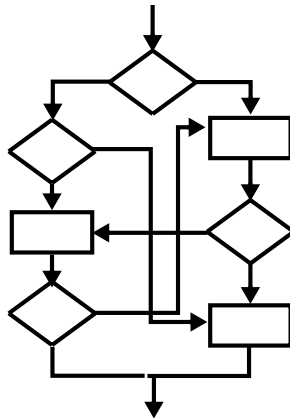
- The first observation is that the loop and conditional constructs (the compound doesn’t need any explicit transfer of control) are much more readable — especially when we handle complex control needs by nesting several such constructs within each other — than **goto**. This doesn’t take much convincing; a simple look at the original structures and their **goto** variants suffices.
- That is not, however, the full story. By sticking to the three mechanisms listed we are restraining ourselves as compared to a programmer who would be using arbitrary **goto** instructions — or, equivalently, arbitrary flowcharts with arrows, possibly crossing each other, from any decision box to any other box. The nickname for such contorted control structures is *spaghetti bowl*; the figure opposite shows an example, still small. The high-level control structures are clearly better for program readability, but that’s only a methodological argument. Could it be that by restricting ourselves to our three control structures and excluding the **goto** we lose something essential? In other words, are there important algorithms that one cannot express without full **goto** power?

The answer, remarkably enough, is **no**. A theorem proved in 1966 by two Italian computer scientists, Corrado Böhm and Giuseppe Jacopini, states that every flowchart of interest in the theory of computation has an equivalent expression using only sequence and loops (the conditional is not even needed).

The transformation rules from arbitrary flowcharts to **goto**-less programs, as derived from their paper, are complex. Without using any such theory you may try your hand — just by using your understanding of the program to spot the implicit loops and rearrange the blocks — at de-**goto**fying a simple structure, or even the flowchart of the last figure.

The implied slander on one of humankind’s most creative culinary inventions is regrettable. On the other hand, real programmers mostly run on cold pizza.

Corrado Böhm and Giuseppe Jacopini: Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM, vol. 9, no. 5, pages 366-371, May 1966. → Exercise 7-E.3, page 196.

Spaghetti bowl

But that is not really what you will do in practice. There is no need to use **gotos** and then remove them. You should build your program directly with the high-level control structures, which have amply proved their adequacy to express algorithms simple and complex in a clear way.

Touch of History: Quashing the goto

Today “Go to” is almost a dirty word in programming, but that wasn’t always so. Once upon a time, branch instructions were the basic control structure. And then without warning appeared in the *Communications of the ACM* of March 1968 — the year, throughout the Western world, of youthful questioning of the established order — an article entitled “*Goto considered harmful*” by Edsger W. Dijkstra. To avoid delaying its publication the editor at the time, Niklaus Wirth, had decided to run it as a “Letter to the Editor”. Through careful reasoning Dijkstra argued that unrestricted branches were detrimental to program quality.

This led to the mother of all programming polemics — then as now, programmers don’t like their habits questioned — which still resurfaces once in a while. But no one would seriously argue any more for unrestricted gotos.

→ See reference & URL in “**FURTHER READING**”, 7.10, page 194.

Dijkstra's short paper, which every programmer must read, beautifully explained the challenge that we face when devising a program:

Touch of the Masters:
Dijkstra on the program and its execution

Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

E.W. Dijkstra, 1968



Dijkstra

No one, then or later, has said this better. A program, even a simple one, is a static view of a wide range of possible dynamic computations, determined by the wide range of possible inputs. So wide indeed is the range — in many cases, potentially infinite — that we can't visualize it; but to ensure the correctness of our program we must somehow infer the dynamic properties from the static view. The discipline of using a nested structure of clear, well-understood mechanisms such as the sequence, the loop and the conditional helps; accepting the unrestricted **goto** would defeat this goal.

Structured programming

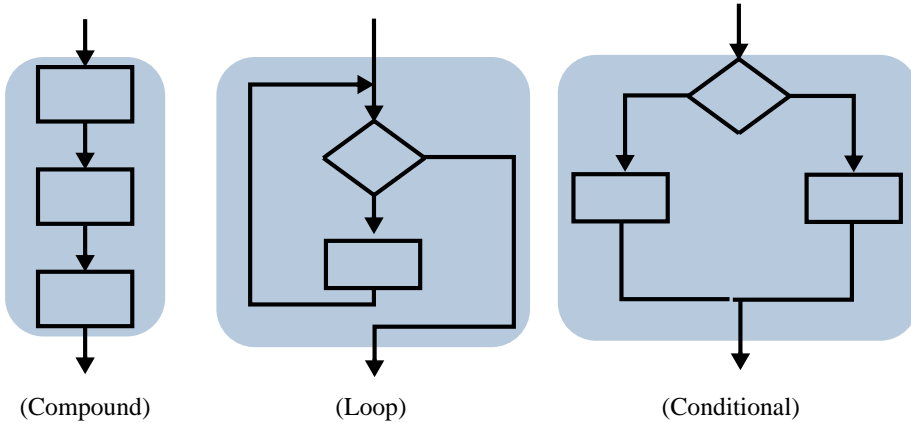
The revolution in views of programming started by Dijkstra's iconoclasm led to a movement known as **structured programming**, which advocated a systematic, rational approach to program construction. Structured programming is the basis for all that has been done since in programming methodology, including object-oriented programming.

As the first book on the topic shows, structured programming is about much more than control structures and the **goto**. Its principal message is that programming should be considered a scientific discipline based on mathematical rigor. (Dijkstra went further, describing programming as “*one of the most difficult branches of applied mathematics*”).

→ “*Structured Programming*”, reference on page [194](#).

What stuck in the mind of the programming masses, however, is the elimination of the **goto** and the restriction of control structures to the three kinds seen in this chapter: sequence, loop and conditional, often called “*the control structures of structured programming*”.

These control structures all have **one-entry, one-exit** flowcharts:



Three kinds of one-entry, one-exit structure

In contrast, arbitrary control structures — see the units of our earlier spaghetti bowl — may have any number of entries and exits. Restricting ourselves to building blocks with one entry and one exit means that we can construct arbitrarily ambitious algorithms through three simple mechanisms:

- **Serial connection:** use the exit of one unit as the entry of another, as an electrical engineer connects the output of a resistance to the input of a capacitor.
- **Nesting:** use a unit as one of the blocks within another.
- **Functional abstraction:** turn a unit, possibly with sub-units, into a routine, also characterized by one-entry, one-exit control flow.

The Böhm-Jacopini theorem tells us that we are not losing any expressive power by limiting ourselves to these mechanisms. The gains in program simplicity and readability — and hence in guaranteeing that the programs are correct, extending them, reusing them — are considerable.

← “*Spaghetti bowl*”, [page 185](#).

The goto puts on a mask

While few people would argue for a return to the general **goto**, the battle for simple control structures is not over. In particular, many programming languages support a form of loop that permits a “break” away from the middle. (There are also break instructions for “multi-branch” variants of the conditional, studied below.) The loop break instruction gives possibilities such as

```

from ... until exit_condition loop
  Some_instructions
  if other_condition then break end
  Other_instructions
end

```

WARNING: Illustration only. Not legal Eiffel.

The effect is that if *other_condition* holds during an execution of the loop body execution terminates prematurely, skipping the *Other_instructions*, any further testing of *exit_condition* and any further iteration.

Other constructs of a similar nature include an instruction **again** that stops the current loop iteration to start the next one immediately.

Such instructions are nothing else than the old **goto** in sheep’s clothing. Treat them the same way as the original:

Touch of Methodology: Sticking to one-entry, one-exit building blocks

Stay away from any “break” or similar control mechanism.

It’s easy to apply the advice to examples such as the above: just rewrite it as

← [“Flowchart for a loop”, page 183.](#)

```

from ... until exit_condition loop
  Some_instructions
  if not other_condition then
    Other_instructions
  end
end

```

Other examples may require more rework but they do not affect the general rule.

The basic argument for that rule is the same one as against the general **goto**: the clarity and simplicity of one-entry, one-exit structures. There’s also a fundamental criterion: our ability to reason about the semantics of programs: in Dijkstra’s terms, to “*shorten the conceptual gap between the static program and the dynamic process*”. With loops as we’ve seen them, a key technique for such reasoning is the **Loop Postcondition Principle**: to understand what a loop does, it suffices to combine the invariant (even if informal) with the exit condition. For example we devised the loop computing the maximum of a set of values to have the invariant

“*max* is the maximum of N_1, N_2, \dots, N_i ”

and the exit condition

$i = n$

making it immediate by visual inspection that the loop — if its initialization does succeed in establishing the invariant and its body in maintaining it, and if it terminates — ensures that *max* is the maximum of N_1, N_2, \dots, N_n . As soon as we introduce **break** instructions or any other way to disrupt the basic control flow of the loop, such reasoning is no longer possible; in fact the very notion of loop invariant goes away, at least in the simple, immediately understandable form we have seen. That’s already reason enough to stick, once again, to the one-entry one-exit scheme.

7.8 OTHER CONTROL STRUCTURES

Sequence, loop, conditional: the threesome of “structured programming” make up the basis of structuring control flow. (By now you might have got the message.) Some variants are interesting too and we’ll now take a quick peek at them.

Since the Böhm-Jacopini theorem tells us that the basic threesome is enough to express all meaningful algorithms, none of the extensions below is *theoretically* necessary; they can all be expressed, in a simple way, as combinations of sequences, loops and conditionals. But that doesn’t automatically disqualify them as useful tool for the programmer, since they might give us a more effective mode of expression in particular cases. Based on this criterion we may divide them into two categories:

C1 • Constructs that provide a welcome improvement over the basic ones, applicable to important practical cases.

C2 • Mechanisms which you need to know since they are present in some common programming languages, but for which no compelling argument exists to justify using them.

This difference is of course partly a matter of opinion, and you'll be able to form your own.

Loop initialization

You may note first that the **from** clause of our loop construct is (as far as I know) specific to Eiffel. As a way to specify the control flow it is of course redundant since instead of

```

from
  Initialization_instructions
until condition loop
  Body
end

```

you may combine the “sequence” control structure with the loop, writing

```

Initialization_instructions
from
  -- Nothing here!
until condition loop
  Body
end

```

This achieves exactly the same effect. The most common loop constructs indeed start at the **until**, or its equivalent in another language.

The reason for including the **from** in the syntax is clear: most loop processes, like most approximation processes, need some kind of initialization, and that initialization is not just some instruction that comes before the loop: we want to treat it as an integral part of the loop, since without it the loop wouldn't work correctly. This is reflected in the loop correctness rules, as the theory assigns a precise role to the initialization: **ensuring the initial validity of the loop invariant**, prior to any iterations of the loop body, each of which (if applicable) must then *preserve* that invariant.

In languages whose loops don't have a **from** clause all you'll be able to do is to write the initialization as a separate compound, perhaps with a comment explaining why it's there.

In Eiffel, this discussion gives us an answer to the question that you may have been asking yourself: if some operations are executed before a loop, should they appear in preceding instructions or in the loop's own **from** clause? Depending on the role of those operations, they might be in either place, or split across the two:

Touch of Methodology:
Where to place pre-loop instructions

If an instruction executed before a loop serves to initialize the loop process, for example to establish its invariant, put it in the loop's **from** clause.

If it is part of a set of operations that simply happen to be executed before the loop in the algorithm of the enclosing routine, put it before the loop.

Other forms of loop

Many programming languages propose a form of loop, usually with the keyword **while**, highlighting the *continuation* rather than exit condition:

```
while Continuation_condition loop
    Body
end
```

WARNING: Sample syntax; not valid Eiffel.

The semantics is: evaluate *Continuation_condition*; if it is false, do nothing; if it is true, execute *Body* and start again. This is equivalent, in our style, to using

```
until not Continuation_condition
```

or **until** *Exit_condition* where *Exit_condition* is the negation of *Continuation_condition*.

The difference is one of viewpoint:

- The **while** form emphasizes **execution**: it reflects that at run time the loop will execute its body as long as the *Continuation_condition* holds.
- The **until** form emphasizes **reasoning** about the program, its correctness and its effect: it reflects that the loop will yield a result that, together with the invariant property, satisfies *Exit_condition*.

Another loop variant shouldn't be confused with **from... until... loop ... end** even though it generally uses the keyword **until** — at the end of the construct rather than the beginning. It may appear as:

```
repeat  
  Body  
until  
  Exit_condition  
end
```

WARNING: Sample syntax; not valid Eiffel.

The semantics is: execute *Body*; then if *Exit_condition* evaluates to true, stop; otherwise start again. Here the *Body* is always executed at least once, whereas the previous variants (**from ... until** and **while**) will not execute it at all if the exit condition is true (or the continuation condition false) on start.

The equivalent of a **repeat** loop in our notation is obvious:

```
from  
  Body  
until  
  Exit_condition  
loop  
  Body  
end
```

This has the disadvantage of repeating the *Body*, whereas we should generally try to avoid code replication. One may counter this criticism by noting that the repetition is part of the algorithm, and that if *Body* contains more than one or two instructions it's appropriate to express it as a routine. Here we reach the realm of opinions. I prefer to have a single loop construct, with its carefully defined semantics and its simple notion of invariant (its counterpart in the **repeat** form is more complicated), and pay the occasional price of repeating a line of code. It is indeed occasional; in practice zero-or-more loops outnumber the one-or-more kind.

Yet another loop variant has the form

```
for i: 1 .. 10 loop  
  Body  
end
```

WARNING: Sample syntax; not valid Eiffel.

with the semantics of executing *Body*, whose instructions generally use *i*, successively for all values of *i* in the given interval: here 1, then 2, and so on up to 10. The boundary values 1 and 10 are just an example, and in some languages can be computed at run time rather than set in the program text.

In the C language and its successors the form is

```
for (i=1; i <= 10; i++) {
    Body
}
```

The first element in parentheses is the initialization of *i*; the second one is the continuation condition; the last is the incrementation operation to be performed after each execution of the *Body*; the notation *i*++ means “increase *i* by one”. Ignoring the very visible differences of syntactical style — rather than keywords, C tends to use many special symbols such as braces, parentheses, semicolons — this implies a fine degree of control over the behavior at execution that is characteristic of this style of programming.

→ Appendix E presents the C language.

The **from** loop of this chapter expresses such schemes simply too:

```
from
    i := 1
until
    i > n
loop
    Body
    i := i + 1
end
```

using the *assignment* instruction *a* := *b* (“Give to *a* the current value of *b*”) studied in detail in a forthcoming chapter.

→ Chapter 9.

This is not the end of the story about the **for** style of loop. One may definitely argue that the **from ... until ... loop** equivalent doesn’t do as good a job of immediately reflecting that the loop is an iteration over a certain interval, 1 .. 10 in our example. This property is buried (as in the C version) in the operations on *i*: initialization, test, incrementation. This is a strong argument for having a higher-level form of loop that simply prescribes: “**apply this operation to all elements of that set**”. But then the **for** style as shown

appears too restrictive: why should we only permit such a scheme for a contiguous interval of integer values? It's easy to think of many other possibilities for *“that set”*. For example, we have already started using lists, such as a Metro line seen as a list of stops or stations; it appears just as desirable that a general mechanism should enable us to ask, in high-level terms, *“apply this operation to all stations on that line”*.

Such a mechanism has a name: **iterator**. We will indeed see that it's possible to define general and powerful iterators, applicable to a wide range of object structures. This will not require any new control structure construct, and will have to wait until the discussion of data structures.

→ *“ITERATING ON DATA STRUCTURES”*, 10.12, page 304

Multi-branch conditional

[To be completed.]

Preview: exception handling

[To be completed.]

7.9 TABLE-DRIVEN CONTROL

[To be completed.]

7.10 FURTHER READING

George Polya: *How to Solve It*, 2nd edition; Princeton University Press, 1957.

The acknowledged reference on becoming better at mathematical problem solving. Don't be put off by the publication date, this book is still a best-seller in its paperback edition.

Edsger W. Dijkstra: *Goto Statement Considered Harmful*, Letter to the Editor, in *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148. Available online at www.acm.org/classics/oct95/.

A famous note that started the programming methodology revolution of the seventies and Structured Programming. Explains why the “Goto” is inappropriate for good programming but, even more importantly, illuminates the process of program construction, concisely (two pages) and effectively. Decades later, still a must-read.

Ole-Johan Dahl, Edsger W. Dijkstra, C.A.R Hoare: *Structured Programming*, Academic Press, 1972.

A classic. Consists of three monographs, the first of which, Dijkstra's *Notes on Structured Programming* is the most famous; but the other two are just as interesting: Hoare's cogent description of the complementary need for *data* structuring, and Dahl's presentation (with Hoare) of the Simula 67 concepts now known as object-oriented programming. Few software books have had comparable influence on the history of the field.

7.11 KEY CONCEPTS LEARNED IN THIS CHAPTER

•

New vocabulary

Algorithm	Branching instruction	Concurrent
Conditional	Conditional branch	Control structure
Cursor	Flowchart	Iterate
Loop	Loop invariant	Loop variant
Overspecification	Parallel	Preserve
Sequence	Unconditional branch	

7-E EXERCISES

7-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

7-E.2 Loops in machine language

Consider a loop of the form

```

from
    Compound_1
until
    i = n
loop
    Compound_2
end

```

Using the machine instructions **BR** and **BEQ** assumed in the discussion of branching, write the corresponding machine-language code.

← [“THE LOWER LEVEL: BRANCHING INSTRUCTIONS”, 7.7, page 180.](#)

7-E.3 Flowchart for a conditional

Following the conventions of the flowchart for a loop, draw a flowchart for the conditional instruction **if Condition then Compound_1 else Compound_2 end.**

← [“Flowchart for a loop”, page 183.](#)

7-E.4 Böhm-Jacopini in practice

Consider the following **goto**-based program extract relying on conditional **goto** instructions:

	<i>Instruction_1</i>
	test c1 goto t3
t2	<i>Instruction_2</i>
t3	<i>Instruction_3</i>
	test c2 goto t2
	<i>Instruction_4</i>

- 1 • Draw the corresponding flowchart.
- 2 • Propose a program extract that has exactly the same run-time effect but uses only compound, loop and conditional as control structures, without any **goto** instruction.

7-E.5 Goto elimination for recursion elimination

(This exercise uses material from a later chapter, chapter [16](#) on recursion.) Express the raw result of recursion elimination for the Tower of Hanoi problem without **goto** instructions. Run the result for a few values of n and check that the results are the same as those of the recursive version.

→ [hanoi_derecursified, page 402.](#)

8

Routines and functional abstraction

The control structures of the previous chapter — compound, loop, conditional and their variants — give us basic mechanisms for scheduling instructions. If they were our only tools, we would always have to express the flow of control in full detail. For complex programs, the depth of nesting would quickly make the structure defy understanding.

To keep that complexity under control we resort to another time-honored problem-solving technique: **identify subproblems**. A subproblem is simply a problem whose solution may help solve other problems. If the subproblem can be solved by providing an element — simple or complex — of the control structure, we can give that solution a name and use it through that name. This is known as *functional abstraction*; the corresponding programming mechanism is known as the *routine*.

8.1 BOTTOM-UP AND TOP-DOWN REASONING

Why can it be useful to identify subproblems? Two complementary answers suggest themselves:

- In solving a problem, we may identify a subproblem to which we already know a solution. Then we'll just plug that solution back into the solution of the larger problem. This is a **bottom-up** use of subproblems: work from what we already know to build up solutions to bigger problems. This style of reasoning is, for example, fundamental in physics and engineering: an engineer will analyze an electrical system and model it in terms of some differential equation of a known type, then use known techniques to solve that equation and deduce properties of the system.

- In other cases we realize that part of a problem by itself constitutes a problem of its own — a subproblem — which we hope will be easier to solve than the overall problem. This may be useful even if we don't already have a solution to the subproblem, because it enables us to deal separately with various parts of the task. You may for example *assume* that there is a solution to the subproblem and use it to solve the larger problem; once you have that larger solution, you will return to the subproblem and take care of its own solution. This is a **top-down** use of subproblems: work on the overall goal, and divide it into a set of smaller goals, to be solved separately. Top-down development is also known as “Divide and conquer” (or “Divide and rule”). We have already encountered a top-down technique: pseudocode, which lets us refer in an informal way to program parts that we wish to expand later. ← Page 110.

Whether in a bottom-up or top-down spirit, the use of subproblems is a form of **abstraction**: ignore the specifics of a particular situation to recognize it as an instance of a general scheme.

In programming, the corresponding construct, capturing the solution to a subproblem, is known as a **routine**.

Touch of Terminology: Routines by any other name

Routines have several other names. You may encounter the synonyms *subprogram* (suitably reminiscent of “subproblem”) and *subroutine*, out of fashion except for the Fortran programming language.

Routines may return a result, and are then called *functions*; a routine that doesn't return a result is called a *procedure*. Both of these terms are, however, sometimes used in reference to routines of the general kind; in particular, C and C-based languages use “function”.

As if this weren't enough, you will also notice, for object-oriented languages, the word *method*, which means the same thing as “routine” but introduces confusions with the usual sense of “method”, as in “*he writes his methods without any method*”.

Routines appear in both bottom-up and top-down development. In their bottom-up role, they support **reuse**: you can take advantage, for your program, of some algorithmic scheme that you or someone else has previously encountered and turned into a routine. In the top-down mode, you can use calls to routines that represent well-identified elements of the processing, and postpone the writing of the routines themselves. This is similar to using pseudocode but more structured, since you have to decide on a precise name and interface for the routine.

8.2 ROUTINES AS FEATURES

A routine captures an algorithm that is applicable to all instances of a class. As such it is one of the two kinds of **feature** of a class. The other kind, yet to be studied, is the *attribute*.

← The definition of “feature” was on page [33](#).

Like any other feature, a routine has:

- A **declaration**, which appears in the text of the feature’s class, and describes all the properties of the routine. The declaration of a routine is also called its **implementation**.
- An **interface**, which retains only a subset of the properties of the routine, those interesting to clients that will use the feature; you can see routine interfaces in the *Contract View* of a class.

← We encountered Contract Views in “What characterizes a metro line”, page [57](#).

We have already encountered many routines, even though we knew them only as features. For example:

- Our very first feature, *explore* in class *PREVIEW*, was already a routine. So is the feature *traverse* that you have been asked to write, under successive variants, in the previous chapter.
- In studying how to use a class through its interface, we used a number of features from class *METRO_STATION*, some of which were routines, such as the command *remove_all_segments* and the query *i_th*. (Some others, such as *sw_end* and *count*, are not routines but attributes.)

← “A CLASS TEXT”, [2.1, page 17](#).

← “COMMANDS”, [4.5, page 63](#) and subsequent sections.

In the first case you had to write the entire routine declaration, but in the second case you only knew the routines through their interfaces, for example:

```
remove_all_segments
-- Remove all stations except the South-West end.
ensure
  only_one_left: count = 1
  both_ends_same: sw_end = ne_end
```

You can see the full routine declaration by looking up the text of the class *METRO_STATION*. You will now learn how to write your own routine declarations.

8.3 ENCAPSULATING A FUNCTIONAL ABSTRACTION

The last [example](#) of our study of conditionals provide a good case for defining a “functional abstraction” in the form of a routine. The overall loop, appearing in the routine *traverse* of our example class *ROUTES*, read:

← Originally on page [178](#), repeated here.

```

from ... invariant ... variant ... until ... loop [7]
  if Line8.item.is_exchange then
    show_blinking_spot (Line8.item.location)
  elseif Line8.item.is_railway_connection then
    show_big_red_spot (Line8.item.location)
  else
    show_spot (Line8.item.location)
  end
  Line8.forth
end

```

Many of the operations apply to *Line8.item*. What’s disturbing is not just the repetition, but the lack of recognition that the Conditional forming the body of the loop is an operation on that object, a Metro station. This will become much clearer if we abstract that operation into a routine. The loop then becomes:

```

from ... invariant ... variant ... until ... loop [8]
  show_station (Line8.item)
  Line8.forth
end

```

relying on a new routine *show_station* whose declaration will appear in the same class *ROUTES*:

```

show_station (s: METRO_STATION) is
  -- Highlight s in a form adapted to its status
  require
    station_exists: s /= Void
  do
    if s.is_exchange then
      show_blinking_spot (s.location)
    elseif s.is_railway_connection then
      show_big_red_spot (s.location)
    else
      show_spot (s.location)
    end
  end
end

```

8.4 ANATOMY OF A ROUTINE DECLARATION

The declaration of *show_station* shows the typical form of a routine. Many elements are already familiar.

A routine is a software element denoting a certain set of operations to be performed on behalf of other software elements, said to **call** the routine. So far we have only one caller to *show_station*: our example loop [8], where the call reads

```
show_station (Line8.item)
```

Such a call may only appear in a routine; here we have assumed that the call is in the routine *traverse* of the same class *ROUTES*. Routine *traverse* is said to be a **caller** of routine *show_station*.

If a routine of a class *C* is a caller of a routine of a class *S*, this makes *C* a *client* of *S*. Here the presence of the call makes *ROUTES* its own client.

← “Client” was defined on page 51.

In the overall system, a routine may have zero, one or more calls, but it always has one declaration, which defines the routine’s algorithm, and appears in a class. Let’s analyze the declaration of *show_station* as it appears on the previous page. The first line

```
show_station (s: METRO_STATION)
```

gives the name of the routine, as well as its **signature**: the list of its **formal arguments**, if any, and their types. Formal arguments represent values on which the routine will operate; each caller will pass these values through **actual arguments**, one for each formal argument.

An actual argument is an expression; its type must match the type of the corresponding formal argument.

The original definition of “argument” covered both formal and actual arguments.

← “FEATURES WITH ARGUMENTS”, 2.4, page 34.

Here the signature involves one formal argument, of type *METRO_STATION*, to which the routine will refer as *s*; in the example call, the actual argument is *Line8.item*. The type of this expression is indeed *METRO_STATION*, since the query *item* of class *METRO_LINE* returns a station. If the types were incompatible, EiffelStudio would produce an error message when you attempt to compile the system:

The screenshot shows a development environment with a code editor and a debugger window. The code editor displays the following code:

```

class
  BAD_TYPE_EXAMPLE
  feature -- Basic operations

  something_else: WRONG_TYPE
  try_bad_type is
    -- Attempt to call feature with wrong argument type
    do
      show_station (something_else)
      -- This would have been the valid call:
      -- show_station (Line8.i_th (5))
    end

  show_station (s: METRO_STATION)
  do

```

Annotations in the code editor point to:

- `WRONG_TYPE`: Type of the actual argument
- `(something_else)`: The actual argument
- `s: METRO_STATION`: Type of the formal argument

The debugger window shows the following error message:

```

Error code: YUAB (2)
Type error: non-conforming actual argument in feature call.
What to do: make sure that type of actual argument conforms to type
of corresponding formal argument.

Class: BAD_TYPE_EXAMPLE
Feature: try_bad_type
Called feature: show_station (s: METRO_STATION) from BAD_TYPE_EXAMPLE
Argument name: s
Argument position: 1
Actual argument type: WRONG_TYPE
Formal argument type: METRO_STATION
Line: 18
do
-> show_station (something_else)

```

An arrow points from the text "Error message with context information" to the error message in the debugger window.

In this example we have passed an actual argument of some arbitrary type *WRONG_TYPE* to a routine that has a formal argument of type *METRO_STATION*. The error message explains what's wrong.

Within the routine *show_station*, we use the formal argument *s* as an expression denoting a station. The operations performed on *s* will, in any call, apply to the corresponding actual argument; in our example call, that's the station denoted by *Line8.item*.

Not all routines have arguments; *remove_all_segments* was an example without.

The remainder of the declaration of *show_station* contains the following elements: ← Page 200.

- Like any feature, a routine should have a header comment explaining what it does.
- There's a precondition, here *s /= Void*, stating that we only want to work on actual arguments that are not void.
- The **do** clause is called the **body** of the routine. It consists of a sequence of instruction — a Compound — defining the algorithm that the routine will execute.
- There could also be a postcondition, although none appears here.

Interface vs implementation

EiffelStudio lets you see both the implementation and the interface of a routine such as *show_station*:

- The implementation (declaration) appears in the default view for the class, known as the “Text View”. It's the declaration of the routine as we have seen it.
- The interface appears if you request the “Contract View” by clicking the corresponding button. Its form, familiar from our earlier study of interfaces of features of class *METRO_LINE*, contains just:

```
show_station (s: METRO_STATION)
  -- Highlight s in a form adapted to its status
require
  station_exists: s /= Void
```

The interface of a routine is intended for programmers of client classes; of the routine's elements listed above, it retains the signature, header comment, precondition and postcondition; but it discards the body, which describes how the routine is implemented. The interface of the routine should only describe *what* the routine does, not *how* it does it. The signature and contracts, complemented by the natural-language explanation of the header comment, suffice to express this “what”.

The Contract View also differs from the Text View by omitting some syntactical details such as the **is** keyword, which are necessary to avoid ambiguity in programs but not required in interface descriptions.

8.5 INFORMATION HIDING

The technique of presenting client programmers with an interface that includes only a subset of the properties of a software element — here a routine, but more generally a class or any other module — is called **information hiding**.

Information hiding is one of the key tools enabling you to build large software systems and cope with their complexity: provide the users of each element with *just what they need* to use it.

In spite of its name, information hiding is not about *preventing* client programmers from seeing the implementation of the mechanisms they use (classes, routines and other features): since the Traffic library and other Eiffel libraries are available in source form, you can use EiffelStudio to peek into the implementations of all the features from *METRO_LINE* and other classes that you have been invited to use through their interfaces. The actual purpose of information hiding is the reverse: *not requiring* the client programmers to look at the implementations when they want to reuse a software element. The amount of information to digest would quickly become enormous. Information hiding enables you to use software by reading only a small part of that information. It's your best ally in the programmer's constant effort to avoid getting swallowed by complexity.

Not all libraries are available in source form; a library supplier may elect to provide interfaces only, usually to preserve proprietary know-how contained in the implementations. Whether to make the implementation available is a commercial or political decision; information hiding is a technical device, unrelated to that decision, for protecting programmers against having to learn heaps of irrelevant details.

Information hiding is a weapon not only against complexity but also against instability. One of the main characteristics of programs as developed in practice is the amount of *change* they must undergo; it's not for nothing that the field is called *software*. Every time some software element changes, it might affect all its clients, potentially triggering a chain reaction of changes throughout the entire system. But if the elements have been well designed, with good choices of what goes into the interface and what remains an implementation decision, many changes will affect the implementation only. Clients will not be affected, since they only rely on the interface. This is an invaluable help to keep software projects under control.

Programming time!

Experimenting with EiffelStudio and information hiding

When you hit the “Compile” button, EiffelStudio doesn’t recompile the entire system, which would take too long. Instead, it recompiles only the classes that you have modified since the last compilation, plus any others that depend on them, directly or indirectly. This is known as *incremental compilation*.

EiffelStudio’s incremental compilation is *automatic*: you don’t need to list the modified classes; EiffelStudio will detect them automatically, and will find out what other classes depend on them.

In this dependency analysis, information hiding is essential: if you change only the implementation of a class, EiffelStudio will spot this, and will not recompile its clients. If your change affects the interface, EiffelStudio will recompile the clients. You may observe this now as follows:

- 5 • Add a routine, say *r*, to *METRO_LINE*. It doesn’t matter what the routine does, but give it an argument and a precondition.
- 6 • In routine *traverse* from *ROUTES*, add a call to *r*. Make sure the call is valid: it must use an argument of the right type.
- 7 • Recompile the system. Notice what classes are being compiled. (Here and in the following you must watch carefully, since the compilation is very fast and displays the names of compiled classes only briefly.)
- 8 • Change an element of the body of *r*, without touching the interface. Recompile, and observe what classes the compilation processes.
- 9 • Now add a precondition clause to *r*; this changes its interface. Recompile, and notice how the compilation processes *ROUTES*.
- 10 • To bring back the system to its previous state, remove *r* from *METRO_LINE* the call to *r* from *traverse*. Recompile and execute to check that everything is back to the previous state.

8.6 PROCEDURES VS FUNCTIONS

There are two kinds of routine:

- A **procedure** performs certain actions; a call to a procedure is, in the calling routine, an *instruction*. The preceding examples, such as *traverse* and *show_station*, are procedures. So are *creation procedures*, studied in an earlier chapter and serving to initialize class instances on creation.
- A **function** computes a certain value or set of values (usually by performing actions too); a call to a function is, in the caller, an *expression*. We haven’t seen any function implementation yet, but several of the features used through their interfaces, for example *i_th* in class *METRO_LINE*, are functions.

← “CREATION PROCEDURES”,
6.5, page 122.

The difference is closely related to one that we already know:

- A procedure implements a *command* feature.
- A function implements a *query* feature.

Commands can only be implemented by procedures, but for queries we will see that there's another implementation: through an *attribute*.

We saw how the signature of a procedure is characterized by a name and an optional list of formal arguments with their types, as in the beginning of the declaration of *show_station*:

```
show_station (s: METRO_STATION) is
... Rest of procedure declaration ...
```

The signature of a function must, in addition, list the type of the value to be returned by the function. We saw this in the interface for functions such as *i_th* in *METRO_LINE*, which returns a result of type *METRO_STATION*, as expressed by the beginning of its declaration:

```
i_th (i: INTEGER): METRO_STATION is
... Rest of function declaration ...
```

The rest of the declaration is the same elements as for a procedure: header comment, pre- and postcondition, **do** clause (body). There's a need, in the body and the postcondition, to denote the value to be returned by the function; this will be through the reserved word **Result**, introduced in the next chapter.

8.7 FUNCTIONAL ABSTRACTION

Routines are the basic algorithmic blocks making up our classes and, through them, our systems.

Use routines as an **algorithmic abstraction** mechanism. To abstract means to concentrate on the essence rather than the circumstances, on the general concept rather than its instances. Abstracting almost always implies **naming**: once you have isolated a useful abstraction, you give it a name for ease of future reference. In programming we encounter two fundamental forms of abstraction:

- **Data abstraction**, which gives us the notion of *class* to describe the abstraction behind our program's data — objects.
- Algorithm abstraction, also called **functional abstraction**, to describe the abstractions behind our algorithms.

In “functional abstraction” the word “function” is taken in opposition to “data”. A better term would be “routine abstraction” since we’ve just seen that functions are technically a more specific notion (routines returning results). The common phrase is, however, “functional abstraction”, and I’ll retain it since you’ve now had enough explanations to avoid any confusion.

To keep your systems manageable even if, overall, their algorithms involve many details, you may rely on routines. Both the bottom-up and top-down views are attractive:

- When you have written an algorithmic element that covers a significant processing step, turning it into a routine enables you to give it a name and a precise specification (signature, header comment and contracts); this turns it into a well-defined software element and, among other benefits, facilitates the later **reuse** of the element. This is the bottom-up view.
- In the top-down view, you may use a routine to capture a step of the processing that you have identified in building a larger algorithm, but for which you haven’t yet written the details, and perhaps do not want to write the details yet as they would detract you from your main goal.

In this second role, routines are often a superior alternative to *pseudocode*. We saw the use of pseudocode, in a top-down development process, to capture elements of the algorithm that you don’t yet want to develop. The example was a pseudocode comment

← [“OVERALL SETUP”, 6.1, page 110.](#)

```
-- “Create line and fill in its stations”
```

which we could replace by a call to a **placeholder routine**, here a procedure *create_fancy_line*. For the system to compile, the routine must exist, even if it does nothing:

```
create_fancy_line is
  -- Create fancy_line and fill in its stations
do
  -- To be completed (your name, today’s date)
ensure
  line_exists: fancy_line /= Void
end
```

Note the postcondition stating that part of the contract: the routine must create an object for *fancy_line*. It would be good to add more clauses..

Touch of Methodology: **Placeholder routines**

If you use a placeholder routine, always include information about *your name* and *today's date*, as well as a full header comment and any other explanation of what you intend to do, so that the purpose doesn't get lost if some time passes before the implementation gets written.

Also, ask yourself if the routine will need any precondition and postcondition, and if so write them from the start, in the placeholder version. They are part of the routine's specification, help you ensure that you understand what you need it for, and will be precious when the time comes to implement it.

8.8 USING ROUTINES

Routines — algorithmic abstraction — are one of your best tools in taming the beast of complexity. Use them generously to capture meaningful algorithmic elements. Use them bottom-up, to prepare existing elements for later reuse; use them top-down, to prepare for elements that you know you will need but don't want yet to write in full.

Programmers concerned with *efficiency*, in particular execution speed, are sometimes wary of using too many routines, since the mechanics of calling a routine almost always means that a call to a routine takes longer than just executing the routine's instructions. A good programmer will, of course, pay attention to efficiency, as to all other qualities of software. But this is seldom a reason to limit the use of routines, for three reasons:

- Modern computer architectures have drastically decreased the time penalty of routine calls.
- Except in the case of a routine call appearing in the body of an “inner loops” executed very many times, any remaining penalty will remain small as compared to the overall execution time of a program performing extensive computations. (If the program does not perform extensive computation, all this doesn't matter anyway.) There will typically be only a few such inner loops, making up a small part of the program, even if they make up a substantial proportion of its execution time. Then you should only worry about these elements, once you have identified them. The rest of the system, where any such small-scale performance consideration have little effect, should be left alone.

- For those program elements where it does matter to avoid the price of a routine call, you don't have to do the job yourself. The EiffelStudio compiler will, in its optimized (“finalization”) mode, perform automatic *routine inlining*. This means that it will expand your routine calls to be executed as if the instructions had been written directly — “in line” — at the place of each call. The advantage of this approach is that you don't have to damage the structure of your program and risk introducing bugs. The inlining process is safe and it is automatic, although you may set some parameters, for example the largest size of routines to be inline.

8.9 AN APPLICATION: PROVING THE UNDECIDABILITY OF THE HALTING PROBLEM

An earlier comment stated that it's impossible to devise an algorithm (“effective procedure”) to determine whether an arbitrary program will halt. Let's prove this result, under the observation that if such a general algorithm existed we could write an Eiffel function that implements it.

← “*Touch of Theory: The Halting Problem and undecidability*”, page 162.

Specifically, we would be able to write a function

```

terminates (root_directory: STRING): BOOLEAN is
    -- Does execution of the system available in root_directory,
    -- if any, terminate?
do
    ... An appropriate algorithm ...
end

```

The argument, *root_directory*, is the name of a directory assumed to contain the system's “Ace” file, that is to say the description of the whole system, giving access to all its classes and specifying the root class and the root creation procedure. We assume for simplicity that the Ace file will be a file called *Ace.ace* in that directory. Being able to solve the Halting Problem then implies that we can complete the “**appropriate algorithm**” so that *terminates* (*r*) will return *True* if and only if there is indeed such an Ace file in *r* and execution of the corresponding system will terminate.

You may change the conventions to include more arguments, or adapt them to another programming language; instead of an Ace file, the argument could simply be the name of a file containing the text of the entire system with all its classes. What matters is that it is possible to pass as argument enough information to let function *terminates* obtain the text of the system whose termination or not it must *decide* (as in “Decision Problem”, Entscheidungsproblem).

All this assumes that the system needs no run-time input. A more general form of the function would handle possible input:

```
terminates_on_input (root_directory: STRING; input: STRING):
                                BOOLEAN is
    -- Does execution of the system available in root_directory,
    -- if any, terminate when applied to input?
do
    ... An appropriate algorithm ...
end
```

We'll use the first form, but the reasoning applies just as well to the second one.

That reasoning is simple. If we had an implementation of *terminates* we could use it to write a one-class system with the following root creation procedure:

```
paradox is
    -- Terminate if and only if not.
do
    from
    until
        not terminates ("C:\your_project")
    loop
    end
end
```

Note empty loop body.

C:\your_project is just an arbitrary directory name; it's a Windows-style directory (folder) name, so on another operating system you'll use something else, for example */usr/home/your_project* on Unix. What matters is that we use the name of the actual directory where we will store the Ace file for our "paradox" system **itself**. Then the call to *terminates* decides whether that system terminates. Now consider what the creation procedure does:

- If the function *terminates* determines by analysis of the system's program text that its execution will **not terminate**, the loop's exit condition **not** *terminates* ("C:\your_project") will be true the first time around, and the loop will **terminate** immediately; so will the entire system since it does nothing else. This is a contradiction.
- If the function determines that execution **terminates**, the exit condition will never be true, so the (empty) loop body will execute forever, and hence the system will **not terminate** — contradiction again.

This suffices to show that it's impossible to write a general-purpose *terminates* function that would ascertain termination for an arbitrary program.

One may devise a more concise version of the argument, ignoring files and directories and instead using Eiffel's "agents". We'll see it in the chapter on agents.

8.10 FURTHER READING

8.11 KEY CONCEPTS LEARNED IN THIS CHAPTER

•

New vocabulary

Actual argument	Body	Data abstraction
Declaration	Formal argument	Function
Functional abstraction	Implementation	Incremental compilation
Information hiding	Placeholder routine	Procedure
Routine	Signature	

8-E EXERCISES

8-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

9

Variables, assignment and references

One of the distinctive features of programs is their use of names, or *entities*, to denote values that may change during program execution, as a result of the computation. The previous examples have implicitly relied on such change, but we haven't yet seen the basic change operation, assignment.

It's a fascinating concept, deceptively simple when you first see it, and full of surprising consequences. We'll study it in this chapter together with a number of related techniques, in particular *references*, which define the run-time object structure.

Math is static, software is dynamic

The ability of a program to change its own environment is the most significant difference between software construction and mathematical reasoning, similar in so many other respects.

Mathematics uses transformations, but they are mechanisms to describe certain values in terms of others, not to change any value that existed before. If I write “*Let $y = \cos(x)$* ”, I am not changing or even creating anything, just giving a name to a value, the cosine of x , that existed all along, whether or not anyone had bothered to talk about it. In particular I am not changing x . Even if after talking about this y I want to contrast the properties of the sine and cosine functions, and continue “*Let's now assume instead that y is $\sin(x)$, then...*”, I am reusing the name y for convenience but talking about another mathematical object. If in describing a sequence I write something like “*Let s_0 be 1, and then $s_{i+1} = 1 / s_i$ for every $i \geq 0$* ” I am referring to an infinite sequence of values, not a value that changes as i increases.

The software perspective is different. We don't just describe results by the properties they must satisfy: we must *compute* them through algorithms whose implementation uses a computer and its memory, large but finite.

The execution of these algorithms proceeds by storing successively computed values in memory. If the memory were infinitely large, infinitely cheap and infinitely fast to access, the execution might choose a different cell for every new value to be stored, such as successive s_i ; but since it's finite we must reuse cells when we don't need their values any more.

So in programming we will have *variables* which, unlike their counterparts in mathematics, deserve their names, as they change value during execution. The presence of such change is one of the major challenges in efforts to reason about programs using the basic tools at our disposal: the tools of logic and, more generally, of mathematics.

9.1 ASSIGNMENT

Assignment is the instruction that allows us to change the value of an entity.

For the examples and exercises of this chapter, you will use a new class called *ASSIGNMENTS*.

Summing travel times

The following simple problem will serve as example: knowing the average time between stops on a Metro line, compute the estimated time for traveling the full line. We will add to *METRO_LINE* a function *total_time* taking care of this.

The principle of the algorithm is straightforward: follow the stops on the line in sequence, and add at each step the time from the previous stop. The basic information comes from a query of class *STOP*:

```
time_to_next: REAL
  -- Estimated travel time to next stop (departure to departure,
  -- except for next-to-last stop: departure to arrival)
require
  has_next: is_linked
```

where *is_linked* tells whether the metro stop is linked to a successor. The type *REAL* is used for the computer approximation of the real numbers of mathematics.

Our desired function *total_time* will have the following general form:

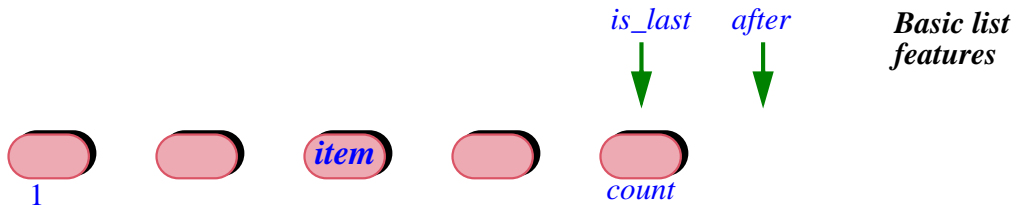
```

total_time: REAL is
  -- Estimated travel time for full line
  do
    from
      start
      -- "Set Result to zero"
    invariant
      -- "The value of Result is the time to travel from first station
      -- to station at cursor position"
    until
      is_last
    loop
      -- "Increase Result by the time to the next station"
      forth
    variant
      count - index
    end
  end
end

```

Result denotes the result to be returned by the function. The two pseudocode instructions will be replaced by assignments.

The boolean-valued function *is_last* tells us whether the cursor is on the last element. Note the difference with the loop schemes of the previous chapter, which stopped on *after* rather than *is_last*.



Quiz time: When to exit from the loop

Why does the loop for *total_time* use *is_last*, rather than the usual *after*, as exit condition?

(Hint: compare the number of stops with the number of intervals between successive stops. Also, compare the variant with the earlier one.)

An Assignment instruction is of the form

```
target := source
```

where *source* is an expression and *target* is a variable entity, such as **Result**. The effect at run time is to change the value denoted by the *target* to the value of the *source*.

We can make good use of this mechanism to complete our function:

```
total_time: REAL is
  -- Estimated travel time for full line
do
  from
    start
    Result := 0.0
  invariant
    -- “The value of Result is the time to travel from first station
    -- to station at cursor position”
  until
    is_last
  loop
    Result := Result + item.time_to_next
    forth
  variant
    count – index
end
end
```

Each time through the loop, we add to the current **Result** the time to the next station. Since we also perform a *forth*, this preserves the invariant. At the end of the loop, that invariant tells us that **Result** denotes the time to travel to the station at cursor position; since *is_last* is now true the cursor is on the last station, so **Result** gives us the total traveling time.

Programming time!

Estimating the time to travel a metro line

Write a function *total_time8* to compute and display the travel time on the Metro Line 8. Use the above model, but to avoid modifying *METRO_LINE* make the function part of *ASSIGNMENTS*, the class for this chapter, adapting it to use *Line8.start* instead of *start*, *Line8.count* instead of *count* etc.

Local variables

In a previous chapter we saw an algorithm scheme for computing the maximum of a set of values. In the absence of assignment it resorted to pseudocode elements of the form ← See e.g. page [161](#).

```
-- "Define max to be  $N_1$ "
-- "Define i to be 1"
-- "Redefine max as the greater of the current maximum and  $N_{i+1}$ "
-- "Increase i by one"
```

We may now express the algorithm fully using assignment. Let's write it as a function that computes the greatest name, alphabetically, of all the station names on a line:

```
highest_name (line: METRO_LINE): STRING is
  -- Alphabetically last of names of stations on line
  require
    line_exists: line /= Void
  local
    i: INTEGER
    new: STRING
  do
    from
      Result := sw_end.name
      i := 1
    invariant ... --- As before
    until
      i = line.count
    loop
      new := i_th (i).name
      if new > Result then
        Result := new
      end
      i := i + 1
    variant ... --- As before
  end
end
```

We have indulged in a little orgy of assignments. The **from** clause initializes **Result** to the name of the first station, *sw_end*, and the integer *i* to one. Then in the loop we find out if the name of the current station, denoted by *new*, is

greater than the name of the current maximum, and if so we replace the value of **Result** by the value of *new* (but otherwise we don't change **Result**, as there is no **else** clause to the **if**). The correctness of this algorithm depends on two properties expressed by the invariants of the corresponding classes:

- A *METRO_LINE* always has at least one station, accessible as *sw_endor*, equivalently, *i_th* (1).
- Every metro station has a non-void *name*.

Also note that order comparison for strings uses alphabetical order: $s2 > s1$ has value *True* if and only if *s2* is after *s1* alphabetically.

Programming time! **Alphabetically highest station name**

Add function *highest_name* to the example class for this chapter, *ASSIGNMENTS*, and use it to display the alphabetically highest name of stations on Line 8 of the Metro.

The principal novelty of this example is its use of *local variables*. The declarations

```

local
  i: INTEGER
  new: STRING
```

introduce two entities, *i* and *new*, which the routine may use for the needs of its algorithm, to store intermediate results. “Local variables” are such entities, local to a routine and introduced by the keyword **local**. You could do without local variables, declaring *i* and *new* (in this example) as features of the class, more precisely *attributes* as studied next. But this would be giving them a status they don't claim: a feature is a property of the class, associated with every one of its instances; here we only need *i* and *new* temporarily for each execution of the routine. When such an execution terminates, *i* and *new* can go away.

You can choose names of local variables freely as long as they don't cause any ambiguity:

Local variable rule

A local variable may not have the same name as a feature of the enclosing class, or as an argument of the enclosing routine.

In principle it would be possible to allow the reuse of feature names as names of local variables, with the convention that within the function the name denotes the local variable; but this would be foolish language design, inviting confusion and errors. Names are cheap; when you need a new entity, choose a new name.

Nothing prevents you, from using the same names for local variables of *different* routines, in the same way that different classes may use the same feature names (some names such as *item*, *count*, *put* ... occur in many different classes). Here there is no ambiguity.

Function results

Result, as used in the last two examples, may appear in a function, where it denotes the result being computed by the function. Remember that *functions* are one of the two kinds of *routine*; procedures, the other kind, can change objects but do not return a result. A function returns a value. **Result** serves to denote, within the function's text, that future result as computed so far. (Obviously, you may not use **Result** in a procedure.)

As a consequence, the instruction (in a routine of class *ASSIGNMENTS*)

```
Console.show (highest_name (Line8))
```

will call *highest_name* and display its value, which is the last value of **Result** as computed by the function's body just before its execution terminates. You will have seen this if you took the last "Programming time".

Result is, formally, a local variable. Its only distinction is that you don't declare it as you do with your own local variables (in declarations of the form *i: INTEGER*); it is automatically available in any function, and implicitly declared for you with the return type you specified for the function — *REAL* for *total_time*, *STRING* for *highest_name*.

This also means that **Result** is a **reserved word** of the language: you may not use it for any of your own identifiers. Reserved words generalize the notion of *keyword*:

← First introduced on page [19](#).

Definition: reserved word

A **reserved word** is an identifier that has a special role in the programming language, and as a consequence may not be used to denote specific elements (such as class names, feature names, entities etc.) of programs.

Keywords are reserved words that play a syntactic role only, as markers. **Result** is an example of a reserved word that's not a keyword because it directly carries a semantic value. Other reserved words in this non-keyword category include **True** and **False**, which also denote values, both boolean.

← "[Boolean values, variables, operators and expressions](#)", page 74.

Entities and variables

Although we haven't seen all kinds of entities yet, it's important to clarify the terminology. We know what an entity is: an identifier that denotes possible run-time values. For some entities, there will be just one such value. Assignment concerns the other kind:

Definitions: variable, variable entity

A **variable entity**, or just **variable**, is an entity whose associated value may change during execution.

Local variables — including **Result** — are variables. The other major kind is attributes, to be studied later in this chapter.

The *target* of any assignment must be a variable.

Swapping two values

Here is a typical use of assignment and local variables. Assume two variables *var1* and *var2* of the same type *T*. The following three instructions will swap their values:

```
swap := var1 ; var1 := var2 ; var2 := swap
```

This requires a third variable, *swap*, typically declared as a local variable of the enclosing routine. It's clear why we need *swap*: we must have a place to store away the value of one of the other two variables before overriding it. A variable such as *swap*, used only for a narrow, immediate purpose, is known as a **temporary variable**.

← Exercise 9-E.3, page 238 asks you to achieve the same result without local variables.

The power of assignment

The symbol for assignment is **:=**; you may read it aloud as “receives”, for example “*i* receives *i* plus one”. The effect, as we've seen, is to replace the value of the *target* by that of the *source* expression.

Some people read it aloud as “becomes”, but “receives” is better: *i* is just *i* and doesn't “becomes” *i* + 1 (except perhaps as in the joke about the impatient German tourist in a London restaurant: “*Vaiter, I vant to become a potato NOW!*”).

“*Bekommen*” in German means “to receive”.

The earlier value of the target is lost — and lost forever: no one’s keeping any record. Assignment is the equivalent, in high-level programming languages, of the basic operation permitted by computers: replace the content of a certain memory cell by a specific value. So if you will need a value again, make sure to record it yourself — through another assignment! — into a variable.

A common scheme in assignments is to use the previous value of a variable in the source of an assignment that has the same variable as its target. This appeared in instructions of both of the routines we’ve seen:

```
Result := Result + item.time_to_next
i := i + 1
```

The goal is to update the value of a variable on the basis of its previous value and new information. That’s very close to the standard mathematical technique of defining a sequence of values, as in the example cited at the beginning of this chapter:

“Let s_0 be given, and then $s_{i+1} = f(s_i)$ for every $i \geq 0$ ”

where f is some function (the example used $f(x) = 1/x$). To compute s_n for some $n \geq 0$ with a computer you may use the loop

```
from
    Result := “The given initial value  $s_0$ ”
    i := 0
invariant
    “Result =  $s_i$ ”
until
    i = n
variant
    n - i
loop
    i := i + 1
    Result :=  $f(\mathbf{Result})$ 
end
```

This scheme is only applicable if you need not retain the successive values, only the last one s_i at each step. Both of our routines used it.

Be sure to remember the difference between the mathematical property $s_{i+1} = f(s_i)$ and the software instruction $x := f(x)$, which reflects the *change* mechanism that’s foreign to mathematics and complicates reasoning about programs. Note in particular the difference between the instruction

```
x := y
```

and the boolean expression

```
x = y
```

as used for example in a Conditional **if** $x = y$ **then** ... The boolean expression has the same properties as an equality in mathematics; it is **descriptive**, presenting a possible property (true or false) of two values x and y . The Assignment instruction is **prescriptive**: it tells the computation to change the value of a variable. In addition it is **destructive**, obliterating the previous value of that variable.

A striking example of the difference is the instruction

```
i := i + 1
```

frequently encountered in loops, using an integer variable i . The boolean expression $i = i + 1$, while legal, would be useless since it always has value false: no integer may be equal to the integer that follows it.

Touch of Syntax: Confusing assignment and equality

The first widely used programming language, Fortran (from the 1950s), used the equality symbol $=$ for assignment. This was clearly an oversight; subsequent languages such as Algol and its successors introduced $:=$ for assignment, reverting $=$ to its standard meaning as equality operator.

For unknown reasons, the C language, in the late sixties, brought back $=$ for assignment, using $==$ for equality. Not only does this convention contradict well-established mathematical properties (for example, $a = b$ in mathematics means the same as $b = a$), but it introduces a frequent source of errors; **if** $(x = y)$... instead of **if** $(x == y)$... is actually legal in C, but has an unexpected effect: assign the value of y to x ; then yield a boolean value (as if the assignment were also a boolean expression), which is *False* if the resulting value of x — the previous value of y — is zero or equivalent, and *True* otherwise! If you use C you must be extremely careful about this source of confusion, which plagues even experienced C developers, and has been documented as the cause of bugs and security attacks in important programs and operating systems.

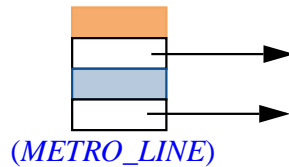
Such recent languages as C++, Java and C# have retained the C convention for assignment and equality, with (in the last two cases) stricter type controls to avoid the bugs mentioned.

9.2 ATTRIBUTES

There are two kinds of variables (entities to which we may assign a value). We have now seen the first kind: local variables, including **Result**. The second, which we'll study now, is *attributes*. It's not completely new: we've seen it implicitly, under the guise of object *fields*, when learning about object creation. But we can now complete our understanding of this concept, and find its place among entities, features and other creatures of our object-oriented bestiary.

Fields, features, queries, functions, attributes

We saw in the discussion of creation that an object, as it exists at run time in the memory of your computer, consists of a number of *fields*, some references, some expanded:



*An object and
its fields*

Like anything other property of the object, these fields must come from the specification of its generating class. Indeed each field comes from a feature of the class, more precisely a query, and even more precisely an attribute.

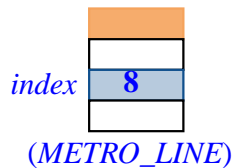
To restart from the beginning, a feature is, as you know, either a command or a query. A query, unlike a command, returns a result. A query can in turn be either a function or an attribute. It's a function if it obtains its result by computing it. For example, in the class *METRO_LINE*:

```
sw_end: METRO_STATION is
  -- End station on South or West side
do
  if not is_empty then
    Result := metro_stops.first.station
  end
end
```

This is a function. On the other hand, we find, in the same class, the following query *without* an algorithm (an **is ... do ... end** part):

```
index: INTEGER
  -- Index of currently considered station in line
```

This is an attribute. Including it in the class means stipulating that every instance of the class will have a field of the given type — *INTEGER* — containing the value of the *index* for the station:



*An object and
its fields*

Assigning to an attribute

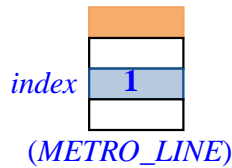
As the comment indicates, *index* in class *METRO_LINE* is the index of the “cursor” position; the cursor is a mechanism allowing clients to explore successive stations of a line by going back and forth. One of the commands for manipulating the cursor is *start*, which sets the cursor to the first station (the one known as *sw_end*):

```
start is
  -- Bring station cursor to first element.
do
  index := 1
ensure
  on_first: index = 1
end
```

A client may call this feature on a particular line, as in

```
Line8.start
```

The effect is to set the value of *index* for the corresponding instance of *METRO_LINE*. If that object previously had its *index* field set to 8, as in the preceding figure, the call will reset it to 1, with no change to other fields:



*“Line” object
after a start*

Line8.start is a qualified call to *start*, from a client. As usual, it’s also possible to call *start* unqualified from another routine of *METRO_LINE*.

Attributes and information hiding

Two other procedures of the class also set *index*:

```

forth is
    -- Move station cursor to next item.
    require
        not_after: not after
    do
        index := index + 1
    ensure
        moved_right: index = old index + 1
    end

go_ith (i: INTEGER) is
    -- Move station cursor to item at position i.
    require
        not_over_left: i >= 0
        not_over_right: i <= count + 1
    do
        index := i
    ensure
        set: index = i
    end

```

All three procedures let clients set the *index* field of any particular *METRO_LINE* object, as in

```
Line8.start
```

```
Line8.forth
```

```
Line8.go_ith (5)
```

[H1]

Just as importantly, such procedure calls are the **only** way for a client to modify this field. You won't be permitted — try it if you wish, and see the compiler message — to write an assignment-like operation

```
Line8.index := 98
```

[H2]

WARNING: syntactically illegal. For discussion only.

This is simply not legal syntax: an assignment may only have a variable entity as its target, and an entity consists of a single identifier; *index* is OK, as in the assignments appearing within the class in procedures *start*, *forth* and *go_ith* shown above, but not *Line8.index*.

The underlying reason is easy to understand. Letting clients directly modify fields in this way would bypass all the safeguards of information hiding and good design. We must remember the general view, illustrated by an earlier picture, of an object as a machine that clients may only manipulate through the operations of its official interface, illustrated as command and query buttons:

← “INFORMATION HIDING”, page 204; see figure page 32.



“*Line*” object as machine

Performing a direct assignment *some_machine.some_field := new_value* would be the software equivalent of unscrewing the casing to reveal the guts of the machine, and starting to rewire the connections with a soldering iron. With an electronic device this would void the warranty; with a software machine, it would void the interface and the associated contracts.

Note in particular how the illegal assignment [2], which at first sight seems equivalent to the procedure call [1], would — if permitted — ignore the precondition of *go_ith* that states

```
require
  not_over_left:  $i \geq 0$ 
  not_over_right:  $i \leq count + 1$ 
```

There's no exception to the rule that any object modification must go through the interface provided by the features of the class. When you write a class, it's both your privilege and your responsibility to decide what you will let clients do to its instances. Given an attribute *a* of the class, of type *T*, you may allow clients to set the corresponding value arbitrarily, by providing a procedure

```
set_a (x: T) is
  -- Set the value of a to x.
  do
     $a := x$ 
  ensure
    set:  $a = x$ 
  end
```

through which clients may use *their_object.set_a (any_value)* without restriction. Or you may introduce a precondition, as in *go_ith*, which restricts the permitted values. Or you might limit clients to more specific ways of setting the value, as would be the case if *METRO_LINE* didn't have *go_ith* but provided only *start* and *forth* as operations that affect the *index* field. Finally, you might decide not to give clients any way at all to modify *index*.

This doesn't prevent you, however, from letting clients **access** the *index* field of an object (rather than modify it). With the class as given, a client may use the expression *Line8.index*; try for example

```
Line8.start
Line8.forth
Console.show (Line8.index)
```

which will display the value 2 in the Console window.

On the other hand you may wish — as a case of full information hiding — to remove an attribute completely from the clients' reach, for access as well as modification. For example *METRO_LINE* has a feature *id_generator* that it uses for its own implementation purposes, and does not make available to

clients in any form. It suffices for the class to include a **feature** clause starting with **feature** {*NONE*}; all the features it introduces are kept away from clients. You can indeed see at the end of *METRO_LINE*, just before the **invariant**, the clause

```
feature {NONE} -- Initialization
  id_generator: ID_GENERATOR
    -- Internal identification for current line
```

This implies that an expression such as *Line8.id_generator* is invalid in any client (try to use it in a class, and see the compiler message). Accordingly, it won't feature in the class documentation as produced by the environment: bring up the Interface View of class *METRO_LINE* now; you won't see any mention of *id_generator*. You may only use this feature, within the class text, unqualified. For example procedure *extend* uses (again check this for yourself) the assignment

```
i := id_generator.generated_id
```

9.3 KINDS OF FEATURE

We have now seen all categories of features; let's go once more over the classification.

The client's view

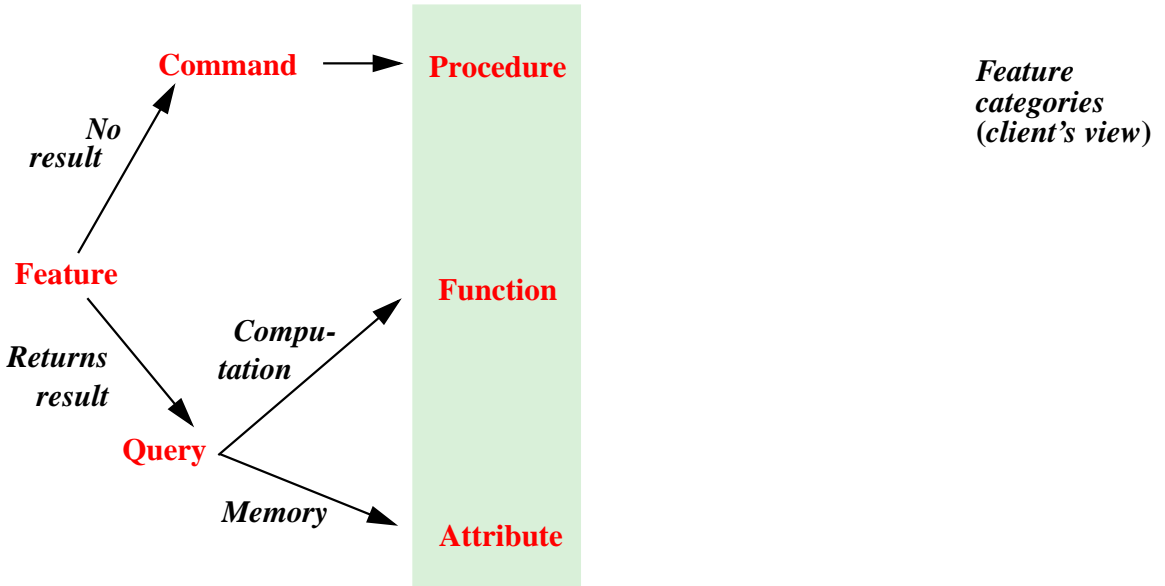
Viewed from the client's perspective, a feature of a class may either:

- Return a result: then it's a **query**.
- Return no result, but be able to modify the target object: then it's a **command**.

In the first case, there are two possibilities depending on how the class author has chosen to implement the query:

- You may choose to *store*, for every instance of the class, the value of the query in one of the instance's fields. This means implementing the query as an **attribute** of the class. It is then the responsibility of every command of the class to update the value of that field if it needs to (for example, executing *forth* affects the value of *index*).
- You may choose instead to *compute* the value of the query whenever requested, using an appropriate algorithm. Then you implement the query as a **function**.

The following figure represents this classification:



“Memory” means that the value is stored, rather than computed. Note that the word “procedure” appears redundant at this stage, being synonymous with “command”.

The notion of **query** is particularly important as a common category for attributes and functions. From the client’s perspective, it doesn’t matter that a query is implemented by storage or by computation. Although the difference between the two categories appears in the class text, it does **not** appear in the class interface. Bring up indeed the Interface View for *METRO_LINE* again; you can see, next to each other, one in the `-- Access` feature clause and the other in `-- Measurement`, the interfaces for

```

index: INTEGER
  -- Index of currently considered station in line
  
```

and

```

count: INTEGER
  -- Number of stations of this line
  
```

But if you now look up these features in the actual class text (not the Interface View) you’ll see that the declaration of *index* appears exactly as shown, since it is an attribute, while the full declaration of *count* reveals it to be a function:

```

count: INTEGER is
    -- Number of stations of this line
do
    Result := metro_stops.count
end

```

Nothing in the Interface View suggests this difference. Internally one is an attribute and the other a function; for the client, they are both just queries.

The policy that treats attributes and functions identically in the Interface View of a class reflects a central principle of software development:

Touch of Methodology: **The Uniform Access Principle**

To clients of a class it must make no logical difference, when they use one of its features, whether the class implements it by storage or computation.

“Storage” is for attributes and “computation” is for functions. “No logical difference” means no difference of functionality; there might still be a difference of execution *efficiency*, as an attribute implementation takes up space, while a function doesn’t but usually requires longer to execute than a simple field access.

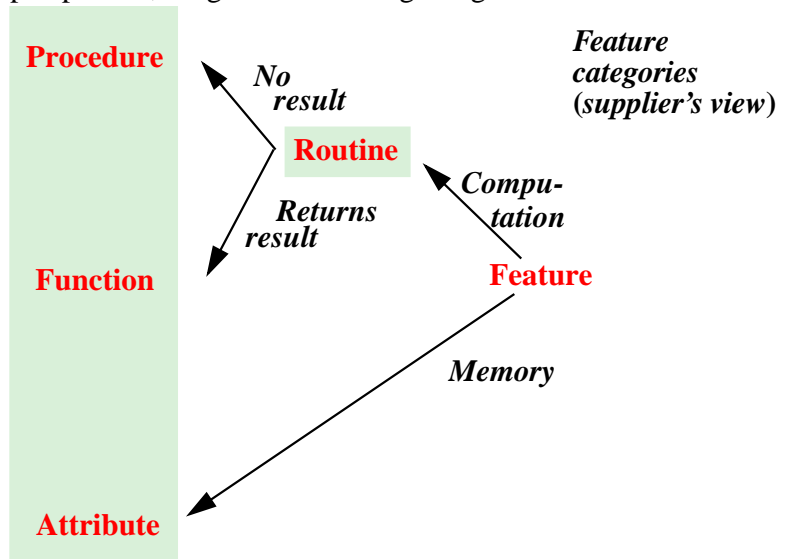
The choice between the two solutions indeed involves space-time tradeoffs, explaining the importance of the Uniform Access Principle: it’s very difficult to know ahead of time what solution will be best; during the course of a project you may have to reverse such decisions several times as a result of time and space measurements. The principle shields client software from the these changes: the notation *some_object.some_query* will remain applicable throughout, so that you may try out various solutions without penalty. If access to attributes and functions used different syntax, you would each time have to update a much larger part of the software than necessary.

The principle further justifies the information hiding policy discussed:

- It is OK to make an attribute available to clients, as in *Line8.index*, especially since we make it available not as an attribute but more generally as a query: the client has no way to know, from the official interface description of the class, whether it’s an attribute or a function.
- It is **not** OK, however, to let clients assign directly to it, as in the illegal *Line8.index := new_value*, since (among other problems) this would reveal it is an attribute.

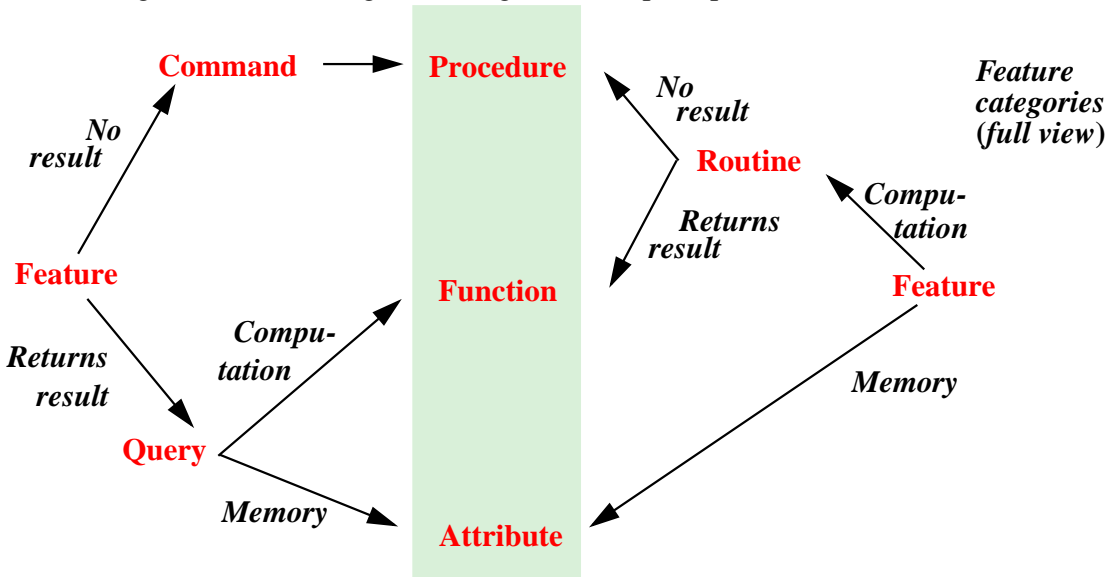
The supplier's view

If we take the viewpoint not of the client but of the supplier class, in other words the implementer's internal perspective, we get the following categories:



The only addition to the previous figure's terminology is the notion of Routine, covering both procedures (the term appears more justified now) and functions.

Putting the two views together, we get the complete picture:



You must know the precise definition of all the terms listed on this figure, and their role in building classes and making them usable by clients.

9.4 ENTITIES REVISITED

To finish with terminology, let's clarify any uncertainty that you may still feel about the fundamental programming concepts introduced so far. Everything regarding *features* should now be clear, but we have also used the terms *entity* and *variable*; it's useful to take a closer look.

Defining entities

Entities were introduced earlier as identifiers denoting possible run-time values; we are now in a position to list all possible variants:

← In "*ENTITIES AND OBJECTS*", 6.2, page 111.

Definition: Kinds of entity

An entity is one of:

- E1 • An attribute.
- E2 • A formal argument of a routine.
- E3 • A local variable of a routine.
- E4 • *Current*, denoting the current object.

So if you were puzzled that *index* from *METRO_LINE* was sometimes referred to as a feature and sometimes as an entity, case E1 is the explanation: *index* is both. In fact *index* is: a feature, and more specifically a query and an attribute; an entity. It's one more thing: a *variable*. Entities are indeed of two kinds:

- Variable entities, or just **variables**, if it's possible for the program to change their values through assignments. This includes local variables (category E3 above) and variable attributes, which we'll study now.
- Constant entities. This includes formal arguments (E3) and *Current* (E4)

Variable and constant attributes

Attributes may be either variable, as in all the examples seen so far, or constant.

Attributes declared in the usual form are variable, for example *index* in

```
index: INTEGER
```

You recognize a **constant attribute** by its declaration including the keyword **is** followed by a value. In *METRO_LINE* you may see (in a **feature** {*NONE*} clause towards the end) the declaration

```
First_id: INTEGER is 1000
```

This introduces the constant integer attribute *First_id*. Note the convention:

Touch of Style: Constants

For names of constant attributes, as for predefined objects, start with an upper-case letter, writing the rest in lower case.

This style is also common for strings, as in

```
Map_title: STRING is "Plan of the Metro"
```

known as a **manifest string**.

Not being variables, constant attributes of any type may not serve as assignment targets: *First_id := 2* or *Map_title := "Something else"* are invalid assignments (try them and watch for the compiler messages).

Constant attributes serve to give names to values that your program may need. You *should* use this technique:

Touch of Methodology: Symbolic Constant Principle

When you need any specific values in a program — other than very simple values such as the integers 0 or 1 to start a loop or increment an index — do not use manifest values directly in the corresponding instructions, but declare constant attributes with these values, and then use these attributes everywhere else.

So if you need a string for an error message, or a physical constant, don't use it directly in the instructions that need them, as in

```
display ("Couldn't send email in allotted time")
```

```
length := 2.54 * length_in_inches
```

WARNING: Not the recommended style.

but declare

```
Timeout_message: STRING is "Couldn't send email in allotted time"
```

```
Inches_to_centimeters: REAL is 2.54
```

and write the instructions as

```
display (Timeout_message)
```

```
length := Inches_to_centimeters * length_in_inches
```

The reason is that such constants (although not the second of these examples) may have to change during the evolution of the program, and then you will want to change just one place; the principle also supports program readability, by encouraging you to give to each constant a name explaining its role in the software. Often, in a large program, you will group such constants in specific classes intended solely for this purpose; for example a particular class may include all the strings appearing in error messages. This facilitates program adaptation and evolution.

Directly using manifest values in instructions is a particularly bad idea for *strings*, as the example error message above (“[Couldn't send ...](#)”), since a successful program may at some point require **localized** versions for various countries. In that case you will use not manifest constants but variables, with the actual strings appearing in external “resource files” rather than in the program.

9.5 REFERENCE ASSIGNMENT

The values that we manipulate — in particular the fields of objects, corresponding to attributes of their generating classes — may be basic values such as integers and booleans, or they may be references. So far we have applied assignment to basic values only; but we also need to assign references. That is in particular how we will build *linked* data structures, such as a list of metro stops where each stop contains a reference to the associated station and link to the next stop on the line.

Implementing the class *STOP* will require such reference assignments. The class interface included the following feature specifications

← See the full specification on page [119](#).

```

set_station (ms: METRO_STATION)
  -- Associate this stop with s.
  require
    station_exists: ms /= Void
  ensure
    station_set: station = s

link (s: STOP)
  -- Make s the next stop on the line.
  ensure
    next_set: next = s

```

indicating that the implementations must set the attributes *station* and *next* respectively. To provide these implementations we need assignment. Here then are the routine texts (not just their interfaces any more):

```

set_station (ms: METRO_STATION) is
  -- Associate this stop with ms.
  require
    station_exists: ms /= Void
  do
    station := ms
  ensure
    station_set: station = ms
  end

link (s: STOP)
  -- Make s the next stop on the line.
  do
    next := s
  ensure
    next_set: next = s
  end

```

A reference assignment **reattaches** the reference to a new object. Previously it may have been void (attached to no object), or attached to another object (or of course to the same object, in which case the assignment changes nothing). To illustrate these possibilities, consider variables *s1* and *s2* of type *STOP* and two creation instructions

```

create s1.set_station (Station_Balard)
create s2.set_station (Station_Issy)

```

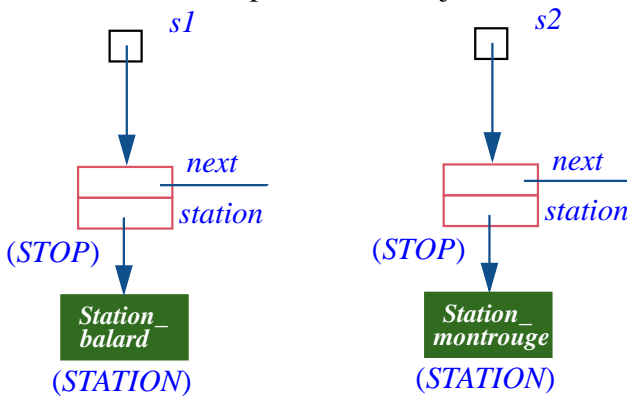
both using *set_station* as creation procedure; this is necessary since we had written the class *STOP* as

← “*CREATION PROCEDURES*”, 6.5, page 122.

```

class STOP create
  set_station
feature
  station: METRO_STATION
  next: STOP
  set_station (s: METRO_STATION) ... As above ...
  link (s: STOP) ... As above ...
invariant
  station_exists: station /= Void
end
  
```

The creation instructions produce two objects:



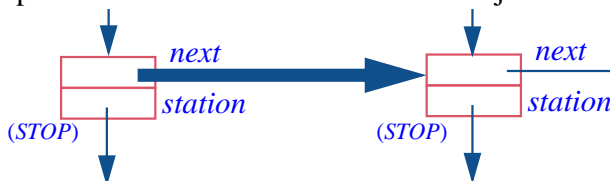
Creating two stops

with *station* references attached (thanks to the creation procedure *set_station*) to two preëxisting *STATION* objects, and *next* references void, since all reference attributes start out void and here *set_station* does nothing about *next*.

To chain the two stops, you may use the instruction

```
s1.link (s2)
```

which updates the *next* reference of the first object



Chaining two stops
(Rest of structure unchanged from previous figure)

as a consequence of the assignment instruction in procedure *link*:

```

link (s: STOP)
  -- Make s the next stop on the line.
  do
    next := s
  ensure
    next_set: next = s
  end

```

This is an example of a reference assignment, which attaches a reference. Here the reference (the *next* field of the first *STOP* object) was initially void, and we assign to it a non-void reference (*s2*), so we attach *next* to an object, the second *STOP* object. We can also use reference assignment to make a reference void, for example by adding the following procedure to *STOP*:

```

make_last
  -- Make this stop the last one on the line.
  do
    next := Void
  ensure
    no_next: next = Void
  end

```

This uses the value *Void*, always denoting a void reference. The following three calls have the same effect (assuming that the value of *v* is void):

```

s1.make_last           s1.link (Void)           s1.link (v)

```

Here is some more playing with references and reference assignments. Let's use the previous example again but with three stations rather than two (the additions are highlighted):

```

create s1.set_station (Station_Balard)
create s2.set_station (Station_Issy)
create s1.set_station (Station_Montrouge)
create s2.set_station (Station_Issy)
s1.link (s2)
s2.link (s3)

```

--- STOPPED HERE ---

9.6 CACHING AND THE SPACE-TIME TRADEOFF

9.7 KEY CONCEPTS LEARNED IN THIS CHAPTER

-
-

New vocabulary

Assignment	Attribute	Local variable	Variable
Temporary variable	Variable entity		

Precise feature terminology

Attribute	Command	Feature	Function
Procedure	Query	Routine	

9-E EXERCISES

9-E.1 Vocabulary

Give a precise definition of each of the seven terms in “Precise feature terminology” above.

9-E.2 Vocabulary

Give a precise definition of all the terms in the above “New vocabulary” list.

9-E.3 Swapping values

Assume variables *var1* and *var2* of type *INTEGER*, with the ordinary arithmetic operations. Can you write instructions that will swap their values, without using any local variables or any other entity? (The answer is an old programming trick; can you think of any limitation?)

← As in “[Swapping two values](#)”, page 220.

9-E.4 Terminology

- 1 • Is every function an entity?
- 2 • Is every function a query?
- 3 • Can a function be a query?
- 4 • Is **Result** an attribute?
- 5 • Is **Result** a feature?
- 6 • Is **Result** an entity?
- 7 • Is **Result** a variable?
- 8 • Are all variables local?
- 9 • Is every attribute an entity?
- 10 • Is every routine a query?
- 11 • Is every query an entity?
- 12 • Is every attribute a variable?
- 13 • Is every function a variable?
- 14 • Is every entity a variable?
- 15 • Can a query be a variable?
- 16 • Can a function be a variable?
- 17 • Is every variable an entity?

10

Fundamental data structures, genericity, and algorithm complexity

On those evenings when it seems you've done nothing all day but store and retrieve things, have a kindred thought for your programs. Many of them — like Traffic with its list-like structures representing metro lines — spend a good deal of their time putting objects into repositories and searching for previously stored objects.

Such a repository (whose elements we will call “items”) is known as a **container**. Lists are only one example, among many kinds differing by the speed of container operations (insert an item, retrieve an item, remove an item, search for items satisfying certain properties, apply an operation to all items...) and the space they require to store the items.

In this chapter we'll study some fundamental container structures, useful across all application areas of computers: arrays, lists of various kinds, hash tables, stacks, queues. This will also be an opportunity to discover three fundamental programming concepts:

- The role of *types* in the development of reliable software.
- Genericity: how to declare type-safe container classes.
- *Algorithm complexity*, a technique to estimate the performance of algorithms and data structures, and naming conventions for features of reusable components.

10.1 STATIC TYPING AND GENERICITY

The first issue container structures raise is a *typing* issue.

Static typing

All the entities of our software are declared with a certain type. This rule enables the compiler to check that any operation you want to apply to an entity x — for example a feature call, $x.f(a)$ — uses a feature that's indeed applicable to it. All the compiler needs to do is to look up the type T with which x has been declared:

- There must be a class for T .
- It must contain a feature f , taking an argument of the right type.

This policy is known as **static typing**: *static* because type properties are specified in the program text, and so can be enforced at compilation time. The alternative, *dynamic* typing, would forsake type declarations, and wait until run time to find out that a feature call $x.f(a)$ tries to apply a feature f to an object that can't handle it.

← “[Definitions: Static, Dynamic](#)”, page 13.

The argument for a static typing policy is twofold:

- *Clarity*: by declaring every entity of the program with a precise type, and every feature with a precise signature, we express the intent behind them and facilitate program reading and maintenance.
- *Reliability*: an invalid feature call is a bug; it's always better to find such mistakes at compilation time than at execution time. The cost of finding and correcting an error increases dramatically the longer you detect it in the software project lifecycle.

Static typing for container classes

How can we apply static typing principles to containers? We're already familiar with lists, since that's what instances of *METRO_LINE* are: lists of instances of class *METRO_STATION*, with features such as

<i>extend</i> (m : <i>METRO_STATION</i>)	-- A command
-- Add m at end of line	
<i>item</i> : <i>METRO_STATION</i>	-- A query
-- Station at current cursor position	

Now let's assume that you want a class *LIST* that can describe lists of *anything*: a list of metro stations, a list of integers, another of objects of any given type. The class should have the above features, but you can't declare the argument m to *extend* and the result of *item* without knowing the type of list items: *METRO_STATION* as above, or *INTEGER* in the second case, or any other type that you have chosen for the objects of a particular list.

We could of course write distinct classes: *LIST_OF_METRO_STATIONS*, *LIST_OF_INTEGERS* and so on. We don't want to do that, since the class texts would be identical except for some type declarations. Such code duplication or quasi-duplication goes against every principle of economy and reuse.

The idea of genericity is to use a single class, here *LIST*, but **parameterize** it so that it can support many types without reprogramming.

Generic classes

Using genericity, we declare class *LIST* as

```
class LIST [G] feature
  extend (m: G) is
    -- Add m at end of line
    do ... end

  item: G
    -- Station at current cursor position

  ...Other features and invariant ...
end
```

G is just a name; it's known as a **formal generic parameter** for the class. ("A" parameter because there may be two or more.) It denotes an arbitrary type, so that within the class we may use it for declarations, as with the argument *m* of *extend* and the result of *item*.

To use the class *LIST* in practice you will declare for example

```
first_1000_prime_numbers: LIST [INTEGER]
stations_visited_today: LIST [METRO_STATION]
```

where each example provides a type, known as an **actual generic parameter** — *INTEGER*, *METRO_STATION* in these examples — to indicate what *G* must represent, within the class, for the particular list.

This technique solves the problem of static typing for general container classes. Assuming the entities

```
some_integer: INTEGER
some_station: METRO_STATION
```

you may use the following valid instructions:

```

first_1000_prime_numbers.extend (some_integer)
stations_visited_today.extend (some_station)

some_integer := first_1000_prime_numbers.item
some_station := stations_visited_today.item

```

This all satisfies the type rules. The formal argument of *extend* in *LIST* is of type *G*; this means *INTEGER* for *first_1000_prime_numbers*, declared as *LIST [INTEGER]*, and *METRO_STATION* for *stations_visited_today*; so it's appropriate to pass as actual argument an integer in the first case and a metro station in the second case. The same applies to the result of *item*.

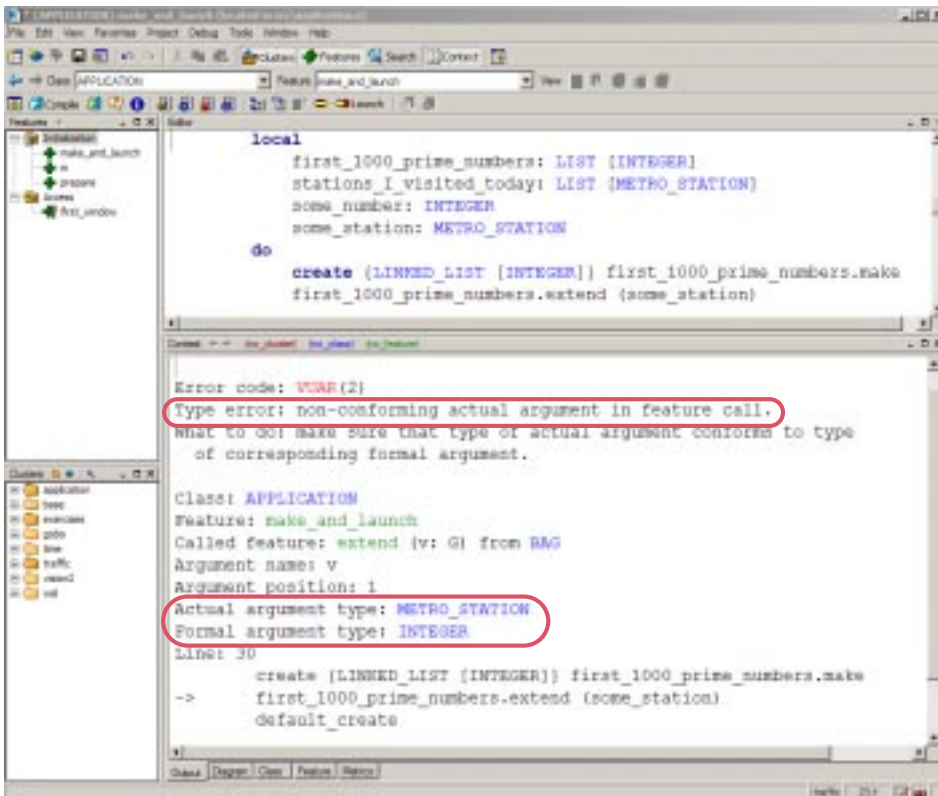
On the other hand, if you try either of

```

first_1000_prime_numbers.extend (some_station)
stations_visited_today.extend (some_integer)

```

you won't get past compilation:



Until the mid-eighties California drivers encountered a single interruption in the entire stretch of Highway 101 from San Francisco through Los Angeles to San Diego: a traffic light in Santa Barbara. This sometimes created the Santa Barbara version of a traffic jam, both for the freeway drivers and for the locals trying to cross over. To those who complained, Jerry Brown, the governor of California in the nineteen-seventies, once replied in a very nineteen-seventies mode that they should instead be grateful for the opportunity to stop and reflect on the deeper meaning of life. This is exactly how you should react to a compilation error such as this one. Do not curse the nasty compiler for preventing you from bringing your program to execution; thank instead the helpful compiler for helping you avert a potential error that could have caused lots of trouble had the program been allowed to proceed to execution.

The techniques just introduced lead to a bit more terminology:

Definitions: Generic class, generic derivation

A **generic class** is class that has one or more generic parameters.

A type obtained by providing actual generic parameters to a generic class is a **generic derivation** of that class.

LIST is a generic class; the type *LIST [INTEGER]*, obtained from *LIST* by providing the generic parameter *INTEGER*, is a generic derivation of *LIST*.

All the container classes studied in this chapter, such as *ARRAY [G]*, *LINKED_LIST [G]*, *HASH_TABLE [G, H]*, are generic. The generic parameter name *G* will always represent the type of items in the container. This is just a convention; you may use for a generic parameter any name that isn't also the name of a class in your system.

Genericity is simply the name of the mechanism allowing classes to have generic parameters, and types, as a result, to be defined by generic derivation.

While we are on terminology: don't confuse the **arguments** of a routine, formal and actual (representing *values* passed to the routine by its callers), and the **parameters** of a generic class, representing *types* allowing a particular use of the class. This distinction is not universally accepted — in the literature you will find “parameter” used for “argument” — but it's important to keep distinct names for distinct concepts.

Validity vs correctness

The goal of the genericity mechanism is, as noted, to ensure the *type validity* of certain kinds of programs (those involving container structures). It's genericity that makes such feature calls as our example *first_1000_prime_numbers.extend(some_integer)* “valid”, meaning that they satisfy the type rules of the language and the compiler will let them through.

This doesn't mean, however, that such instructions will always work correctly. The target of the call, *first_1000_prime_numbers*, might be void at execution time; or *extend* might have a precondition that *some_integer* does not satisfy. We have two different notions at play:

Definitions: Validity, correctness

A program is **valid** if it satisfies all the type rules and other static consistency rules of the language, guaranteeing that certain kinds of run-time malfunctions may never happen.

A valid program is **correct** if it will always execute in accordance with the desired behavior, and never cause a contract violation or other run-time malfunctions leading to failure of the execution.

The second definition of correctness applies only to valid programs. Indeed, for a *statically typed* language (a language with precise type rules, such as Eiffel) it makes no sense to ask about correctness unless the program has passed validity checks.

Examples of the “certain kinds of run-time malfunctions” ruled out by validity include the application of a feature to an object that can't handle it.

Why do we need two notions? It would be good if validity implied correctness, so that once your program has passed muster with the compiler you can go home and rest assured that it will execute properly. Dream on. Although programming languages have been getting better and better at defining static rules that catch errors at compile time, there remain cases that can only be detected during execution. For these, other mechanisms are available, such as *exception handling*.

Devising a framework in which validity implies correctness is an old quest, the Philosopher's Stone of programming. For the moment, we have to accept that they remain distinct; but you'll quickly find out that static typing, especially when combined with techniques of Design by Contract, gives you an outstanding tool to kill bugs before they've even had a chance to attack you.

Classes vs types

With genericity we may take a closer look at the relationship between the notions of class and type.

A type is the description of a set of run-time values: type *INTEGER* specifies the properties of integers as they will be used in your programs, type *METRO_STATION* describes the properties of run-time objects representing stations, and so on.

A class is a program module defining a collection of features (and their properties, such as the class invariant) applicable to a set of run-time objects.

The connection is very close: the “set of run-time objects” associated with a *class* is, at least if the class is not generic, a *type*. Any non-generic class such as *INTEGER* and *METRO_STATION* is indeed a type, and can be used in type declarations for entities, as in the examples used earlier:

```
some_integer: INTEGER
some_station: METRO_STATION
```

This use of classes as both the basic units (“modules”) of program texts, a *static* notion, and the typing mechanism for objects, a *dynamic* notion, is central to the object-oriented style of programming, which would better be called *class-oriented*.

The connection between classes and types remains just as strong with genericity. What’s new is that a class such as *LIST* or *ARRAY* no longer immediately gives us a type; it gives us a template for a type. To get an actual type, it suffices to perform a generic derivation by providing an actual generic parameter. For example:

- *INTEGER* and *METRO_STATION* are both classes and types. This is true of any non-generic class.
- *LIST* and *ARRAY* are classes; *LIST [METRO_STATION]*, *ARRAY [INTEGER]* and so on are types. This is applicable to any generic class.

We can turn this observation into a precise definition:

Definition: Class type

A **class type** is one of:

- T1 • A non-generic class.
- T2 • A *generic derivation* of a generic class (that is to say, the class name followed by appropriate actual generic parameters). In this case the class type is said to be *generically derived*.

The notion is “class type” rather than “type” in general since there are a few other kinds, although class types are the most important.

Nesting generic derivations

There remains to clarify what you may use as an *actual generic parameter* for a generic class. The answer is easy to guess: a type. You can see this in the last examples: in `LIST [METRO_STATION]` the actual generic parameter `METRO_STATION` is a type; so is `INTEGER` in `ARRAY [INTEGER]`.

Perhaps you're sensing something strange in these definitions:

- We've just defined types by stating (clause [T2](#)) that they may be obtained from a class and actual generic parameters.
- Now we're defining an actual generic parameter as a type.

Is this a worthless circular definition? No. It's just an example of a *recursive* definition, one that builds new elements of a set under definition — here the set of types — by using elements previously obtained under the same definition. The process is clear:

- Through clause [T1](#) of the definition we know that, for example, `METRO_STATION`, a non-generic class, is a type.
- We may now use clause [T2](#) to deduce that `ARRAY [METRO_STATION]` is also a type.

Recursion is a fascinating technique, not just for such concept definitions but for routines and data structures. We'll have a full [chapter](#) devoted to it, but this example should suffice to show that, in the present case, there's nothing inconsistent or strange in the recursive definition of "type".

→ Chapter 16, in particular "[Recursive definitions](#)", page 360 and "[Bottom-up interpretation of a construct definition](#)", page 392.

The definition in fact opens interesting possibilities. The type used as actual generic parameter in [T2](#) can follow not just from [T1](#) but also again from [T2](#); in other words it can be generically derived. This allows types such as

```
LIST [LIST [INTEGER]]
LIST [ARRAY [METRO_STATION]]
ARRAY [ARRAY [ARRAY [INTEGER]]]
```

and so on without limitations. This is not just a theoretically pleasant possibility, but a practical mechanism, as we'll encounter the need for lists of lists, lists of arrays and other multi-level containers.

10.2 CONTAINER OPERATIONS

The container structures studied in the second part of this chapter, all of common use in programming, are quite diverse, from arrays, linked lists and other kind of lists to hash tables and tree structures.

Why such diversity? It's due to the difficulty of providing equally efficient implementations for the many operations that we need to execute on containers. For example arrays let you very quickly get to an item if you know its index; but they are not good — read: too slow — for insertion of new items. Linked lists are reasonable for insertion, but slower than arrays for index-based access. When you need a container, you'll have to choose one of the available structures depending on the operations you require from it.

For our study of fundamental container variants we must first define these fundamental operations. We'll look first at queries, then at commands. G will denote the type of the container's items; it's the generic parameter of the corresponding classes, as in `ARRAY [G]` or `LINKED_LIST [G]`.

Queries

We'll need to find out if a container is empty (has no items). The query, returning a `BOOLEAN`, will be called `is_empty`. Its signature is just

```
is_empty: BOOLEAN
```

in other words it takes no argument but is called under the form `c.is_empty`, yielding a boolean value for any container `c`.

To find out if a particular item appears in a container we'll use

```
has (v: G): BOOLEAN
```

To find out how many items are in a container:

```
count: INTEGER
```

An invariant clause, applicable to all relevant container classes, states:

```
is_empty = (count = 0)
```

To obtain an item from the container — *any* item, chosen by the container’s policy, not by the client:

```
item: G
```

Some containers let you obtain an item given by an integer index, as in “give me the third item”. The query is:

```
item (i: INTEGER): G
```

Using the same name as for the previous operation causes no ambiguity because the signature is different; no structure reviewed here has both variants.

An integer is only a special case of a **key** enabling you to retrieve an item from some information associated with it. There are many different kinds of keys; one of the most common is a **string**, as in a container representing a Web page and allowing a search engine to ask whether certain words appear. For string keys the query will be

```
item (i: STRING): G
```

We’ll learn how to generalize the key type. Here too reusing the name *item* causes no confusion.

Commands

The creation procedure that sets up a container will usually be called *make*. Often it has no argument, but sometimes it takes one indicating an expected number of items:

```
make (n: INTEGER)
```

For all the containers of this chapter *n* is only an indication to guide the initial creation of the data structure, not an absolute maximum.

The most common operation for adding or replacing an item is called *put*, with one of the following signatures, matching one of the signatures for *item* but with one more argument indicating the new value:

```
put (v: G)  
put (v: G; i: INTEGER)  
put (v: G; k: STRING)
```

The postcondition should always include the clause

```
inserted: has (x)
```

and, in addition, should express the relationship with the corresponding version of *item*:

- *item* = *v* if *put* has no argument (the first case).
- *item* (*i*) = *v* for the version with an integer index.
- *item* (*k*) = *v* in the last case.

The procedure *put*, when present, may either *add* an item or *replace* an existing one. Sometimes we need to distinguish, using one of

```
extend (v: G)
extend (v: G; i: INTEGER)
extend (v: G; k: STRING)
```

with the postcondition

```
one_more: count = old count + 1
```

or one of

```
replace (v: G)
replace (v: G; i: INTEGER)
replace (v: G; key: STRING)
```

with

```
same_count: count = old count + 1
```

When either of *extend* and *replace* exists, *put* is usually a synonym for one of them, corresponding (if both are present) to the more common use. In all cases, the postcondition clause *has* (*v*) expresses that after you've added an item the structure must answer "yes" if asked about its presence.

The procedure to remove an item is, depending on the context, called *remove* or *prune*.

Automatic resizing

We saw above that creation procedures (usually *make*) that specify an initial size always mean it as an indication, not a permanent limit. The data structures of EiffelBase (Eiffel's container library) are almost all either unbounded or, if they have an initial bound, resizable. Part of what defines a good programmer is, indeed, avoidance of absolute limits.

Don't let anyone lock you — and the users of your programs — in a fixed box. Design your data structures, like EiffelBase, so that if the size of the data exceeds expectations they don't give up (as some libraries do) but just resize their implementation whenever possible. Computers have big memories; there are few more stupid program events than hitting a fixed limit and not being able to reallocate the structure. As we'll see, even arrays (often treated elsewhere as fixed-size structures) are resizable in Eiffel.

It so happened that just during the writing of this chapter the world was fixated on the initially unsuccessful exploration of Mars by the NASA's Spirit and Opportunity rovers. Spirit was silent for more than a day, rebooting again and again. Engineers suspected all kinds of possible equipment failures, until it surfaced that it was a software issue: the system had room for a fixed number of file handles, and needed more files than planned. No one had apparently thought of using Eiffel and its resizable structures.

See www.newscientist.com/news/news.jsp?id=ns99994610.

One year later, a few weeks after the US elections, it transpired the vote-counting software in the San Francisco Supervisor vote had failed because of “a *hard-coded constant maximum number of voters that was set too low*”.

Peter G. Neumann, “Some 2004 voting anomalies”, catless.ncl.ac.uk/Risks/23.59.html#subj2

Please don't fall into such pitfalls:

Touch of Methodology: Don't box in your users

Don't use constant built-in limits. Let your data structures resize themselves to adapt to the size of each instance of the problem.

Standardizing feature names for basic operations

The names cited above recur throughout the libraries. Even a casual look at container classes through EiffelStudio will show that most of them have features called *item*, *has*, *put* etc.

This is a deliberate choice. One could of course invent new names for each class, reflecting the specific properties of the corresponding kind of container. But these peculiarities are already captured by the signature, header comments and contracts of the features, for example in *put* for *ARRAY*

```
put (v: like item; i: INTEGER)
  -- Replace i-th entry, if in index interval, by v.
require
  valid_key: valid_index (k)
ensure
  replaced: item (i) = v
```

and in *put* for *STACK*:

```
put (v: G)
    -- Push v onto top.
require
    extendible: extendible
ensure
    pushed: item = v
```

so that no confusion can result. Using consistent terminology facilitates the library's ease of use and ease of learning: when discovering a new class, you can quickly identify the key features and their purpose.

Touch of Methodology:
Standard feature names

It is a good idea to use the standard names, when applicable, for features of your own classes, enhancing their consistency and readability.

10.3 ESTIMATING ALGORITHM COMPLEXITY

The various container data structures will differ by how much space they use to store items, and how much time they require to implement the essential operations as just reviewed.

We need a reliable way to contrast such performance for various data structure choices. It's not enough to measure concrete performance on specific examples and report that “*on average item took 10 nanoseconds for arrays and 40 nanoseconds for linked lists*”:

- To talk about averages we must have a significant statistical distribution; there is no clear way of determining such a distribution for container sizes (how many 10-item containers, how many with 1000 etc.).
- You can't easily infer from the measurements how the results will scale up. Some techniques can be very good for small structures, but what matters in performance-critical applications is how well they do for large sizes. (For 1000 items, almost any container will do a decent job.)
- The result is closely tied to the context of the measurements: machine, operating system, even programming language. It is not rare to see the same experiment give radically different outcomes on different machines.

The accepted measure of algorithm complexity, known as **abstract complexity** — also familiarly as “Big-O notation”, sometimes written “Big-Oh” to emphasize that the O is a letter —, provides a estimate freed from such contingencies.

Measuring orders of magnitude

Abstract complexity relies on two principles:

- Provide the measure as a function of the *size* of the data structures under consideration. For most of the examples in this chapter it’s a single parameter: *count*, the number of items in a container.
- Define the function not by an exact formula but by an *order of magnitude*, the O in “Big-O”, as in $O(\textit{count})$ (pronounced “ O of *count*”).

When we say that the time for a search operation in a list of *count* elements is $O(\textit{count})$ we mean that for large values of *count* it grows at most **proportionally to *count***. Another operation may be $O(\textit{count}^2)$, meaning that its execution time grows at most proportionally to the square of the number of elements. The same conventions are used for estimating space requirements.

In such a measure:

- Constant multiplicative factors do not matter: $O(100 * \textit{count}^2)$ means the same as $O(\textit{count}^2)$. The justification for this convention is that we can’t attach any long-term importance to multiplication by any constant, since the same algorithm implementation may become 100 times faster or slower just by being moved to a different machine; but how its computation time varies when *count* grows doesn’t depend on such technical choices.
- Constant additive factors also do not matter: $O(\textit{count}^2 + 10000)$ means the same as $O(\textit{count}^2)$. The constant may have a strong influence for small *count*, but as *count* grows it fades away.
- Similarly, any additive factor with a smaller exponent doesn’t matter: $O(\textit{count}^3 + \textit{count}^2)$ is the same as $O(\textit{count}^3)$.

As a consequence, to express that an algorithm takes constant time — or, more realistically since several executions are unlikely to take exactly the same time, that its execution time is *bounded* by a constant — we say that it’s $O(1)$. We might just as well say $O(10)$ or $O(1000)$, but 1 is the convention.

Mathematical basis

The Big-O notation may seem informal, but it's possible to define it in a completely rigorous way, as a relation between two functions:

Definition: Big-O notation for abstract complexity

Let f and g be two functions from natural numbers to non-negative real numbers. Function f is said to be $\mathbf{O}(g)$ (or, more commonly, $f(n)$ to be $\mathbf{O}(g(n))$, spelling out the argument) if there exists a constant K such that $f(n) / g(n) < K$ for every natural number n .

An algorithm is $\mathbf{O}(g(n))$ in time or in space if the function giving its execution time or space occupation in terms of the input size n is $\mathbf{O}(g(n))$.

When reading analyses of algorithm complexity you may encounter statements such as " $f(n) = g(n) + \mathbf{O}(n^2)$ " as an abbreviation for " $f(n) = g(n) + s(n)$ for some function s , where $s(n)$ is $\mathbf{O}(n^2)$ ". The intent is to state that f is "like" g except for a term in $\mathbf{O}(n^2)$.

As a result of the definition, if a function is $\mathbf{O}(n^2)$ it is also $\mathbf{O}(n^3)$, $\mathbf{O}(n^4)$ and so on. This is because the Big-O specification gives an upper bound, not a precise estimate.

Logarithms frequently arise in the analysis of algorithm complexity. For example the best algorithms for *sorting* a list of n values are $\mathbf{O}(n * (\log n))$. Such a formula doesn't specify the logarithm base (such as 2 or 10), because a change of base only contributes a multiplicative constant, per the formula $\log_b n = \log_b a * \log_a n$.

Making the best use of your lottery winnings

This convention of ignoring multiplicative constants can be surprising at first. If an algorithm takes *count*^{1.5} nanoseconds, abstract complexity considers it less good than one taking $10^6 * \textit{count}$ nanoseconds, even though the second one runs faster for up to one million items. What the convention gives us is an understanding of the essential behavior of algorithms as a function of the growth of the problem size.

The following observation helps understand the benefit. Consider four algorithms with performance such that the biggest problem size they can tackle in 24 hours of continuous operation on your computer is respectively N_1 , N_2 , N_3 , N_4 . Their abstract complexities are $\mathbf{O}(n)$, $\mathbf{O}(n \log n)$, $\mathbf{O}(n^2)$, $\mathbf{O}(2^n)$. You win at the lottery and have the opportunity to buy a computer that's one thousand times faster than your current one. What does this get you?

Adapted from Aho, Hopcroft, Ullman; see "FURTHER READING", page 306.

- With an $O(n)$ algorithm you can now solve a problem that's a thousand times bigger: $1000 * N_1$.
- With $O(n \log n)$ the improvement is still multiplicative, with a factor that is close to 1000 for large N_2 .
- With $O(n^2)$ you multiply the maximum problem size by a factor of about 32 (square root of 1000).
- With $O(2^n)$ your new dream machine *increases* N_4 — it's an addition, not a multiplication! — by just 10.

This question — how big a problem you can solve in a given time, rather than how much time it will take to solve a problem of a given size — is often the right way of looking at efficiency issues.

Consider for example a next-day weather forecast program. (Meteorology has made spectacular progress in the past two decades thanks to computer modeling.) The program works from past data collected at a number of points on a geographical grid. More grid points means more accurate predictions. To assess the program's efficiency, the useful criterion is not how long it takes to process a fixed number of grid points, since an outstanding next-day forecast won't help if it takes 48 hours to complete. It's the reverse question: how many data points you can process in a fixed time, for example one hour.

This reasoning illustrates what abstract complexity gives us: a view of algorithm efficiency free from superficial technology considerations, but helping to understand the benefits of potential technology improvements.

Abstract complexity in practice

When measuring the Big-O complexity of an algorithm you may be interested in any of three variants, and should clarify which one you report:

- **Average complexity**, assessing the average time or space taken up by the algorithm. As already noted this is only meaningful if we have a probability distribution on the algorithm's input; usually the distribution considers all possible inputs equally likely.
- **Maximum complexity**, also called **worst-case** complexity. assessing the time or space required by the inputs that make this measure highest.
- **Minimum** or **best-case** complexity, less often useful in practice (other than for programmers who believe in the Tooth Fairy), but sometimes interesting for purposes of comparison.

Presenting data structures

In the remainder of this chapter we look at fundamental structures. The presentation relies on the **EiffelBase** library of data structures and algorithms, which provides reusable classes for all the concepts under study: *ARRAY*, *LINKED_LIST*, *HASH_TABLE*, *STACK* and so on.

The description takes the viewpoint of the *client programmer* (also known as “you”): someone who will take advantage of these library classes to write a new application that uses arrays, linked lists etc. Features will, as a result, be introduced through their **contract forms**.

The presentation explains basic implementation techniques but does not, as a rule, show feature implementations. That would be the role of a companion book (which I hope someone will write) focusing on the fascinating topic of data structures and algorithms. You *can* see implementations if you wish: EiffelBase is open source software, included with any delivery of EiffelStudio, and you are welcome to explore its code, written with the explicit goal of serving as a model of O-O style, and refined again and again over the years.

Not to imply it's perfect ...

10.4 ARRAYS

We start with one of the most commonly useful container types, arrays.

Arrays are a software notion, but their importance comes from a hardware property: the addressing mode of the type of main memory used in today's computers, known as *Random Access Memory*, or just “Random Memory”. In spite of the name this doesn't mean that the computer throws a dice to decide which cell to access (interesting idea, though) but that the time to access a memory cell — either to read it or to modify it — doesn't depend on the cell's address. (Understand “random” as in “*you can pick an address at random and not worry about the effect on access time*”.) If you have a 2 GB memory, it won't make any difference whether the cell is the first (address 0), the last (address $2^{33}-1$) or anywhere in-between.

Random access memory stands in contrast to *sequential access* memory, where you access an item by first traversing a set of preceding elements. Magnetic tapes are a typical example: the tape head reaches a particular position by rolling the tape to that position. You may also think of analogies in non-computer devices:

(Sequential)

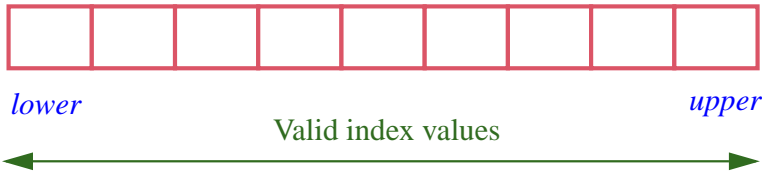


(Random)



Sequential and random access

Arrays take advantage of the random access property by letting you define and manipulate structures made of a number of items stored in contiguous memory locations, and each identified by an index:



An array

Bounds and indexes

An array has a lower bound and an upper bound, given in the class `ARRAY[G]` by the queries

```
lower: INTEGER
    -- Minimum index
upper: INTEGER
    -- Maximum index
```

The invariant of the class implies that *count*, the number of items (also accessible as *capacity*), is $upper - lower + 1$. Since $count \geq 0$, we must have

```
lower <= upper + 1
```

The case $lower = upper$ corresponds to an array with one element; $lower = upper + 1$ corresponds to an empty array (you can visualize it, on the last figure, as *lower* moving right and *upper* moving left until they cross). This is a legitimate state for an array:

Touch of Methodology: **Extreme Cases Principle**

When designing object structures, for example containers, consider extreme cases — empty structure, “full” structure if there’s a maximum capacity — and make sure that the definition still makes sense for them.

There's a long history of bugs due to inadequate handling of extreme cases. People will think of (and test for) cases in which an array or other structure has items; then in an execution for some particular input data, the container happens to be empty, and everything blows up. Following the above advice will avoid such nasty problems.

The class invariant is your primary guide to checking that the definition “still makes sense”. Here the case $lower = upper + 1$ remains compatible with the invariant clause $lower \leq upper + 1$; it's the smallest value of $upper - lower + 1$ (that is to say, *count*) that satisfies this requirement.

To access and modify array items, you'll use integer indexes. A query is available to find out if an integer is a meaningful index:

```
valid_index (i: INTEGER): BOOLEAN
  -- Is i within array bounds?
ensure
  Result implies ((i >= lower) and (i <= upper))
```

Creating an array

To create an array, you'll provide the desired lower and upper bounds:

```
your_array: ARRAY [SOME_TYPE]
...
create your_array.make (your_lower_bound, your_upper_bound)
```

using the creation procedure

```
make (min_index, max_index: INTEGER)
  -- Allocate array; set index interval to min_index .. max_index;
  -- set all values to default.
  -- (Make array empty if min_index = max_index + 1).
require
  valid_bounds: min_index <= max_index + 1
ensure
  lower_set: lower = min_index
  upper_set: upper = max_index
  items_set: all_default
```

As the first two postcondition clauses indicate, the procedure sets *lower* and *upper* to the given values, *your_lower_bound* and *your_upper_bound* in the example. These are arbitrary expressions; you can use constants, as in

```
create yearly_twentieth_century_revenue.make (1901, 2000)
```

where the bounds are set in the program text; but you may also use variables and more general expressions, as in

```
create another_array.make (m, m + n)
```

In examples such as these the index interval has a meaning of its own, such as directly representing the years of the 20th century. If you just want a sequence of n values that can start anywhere, the common convention is to use the bounds 1 and n :

```
create simple_array.make (1, n)
```

The C language and its successors (C++, Java, C#) require all arrays to start their indexes at 0. In examples such as “years of the 20th century” this means that you’ll have to perform back-and-forth translations (here adding or subtracting 1901) between the physical index and its intended meaning. For cases such as *simple_array* the choice of 0 or 1 as starting index is partly a matter of taste; I definitely prefer 1 because with the 0 convention the last item of an array of size n has index $n-1$, a frequent source of errors.

The query *all_default*, in the last postcondition clause of *make*, expresses that all items of an array of type *ARRAY [SOME_TYPE]* will, on creation, be set to the default value for *SOME_TYPE*: zero for *INTEGER* and *REAL*, false for booleans, void reference for any reference type.

Accessing and modifying array items

The basic query and command to obtain and modify an array item are:

```
item (i: INTEGER): G
  -- Entry at index i, if in index interval
require
  valid_key: valid_index (i)
```

```
put (v: like item; i: INTEGER)
  -- Replace i-th entry, if in index interval, by v.
require
  valid_key: valid_index (i)
ensure
  inserted: item (i) = v
```

→ The declaration of *item* also contains **alias** and **assign** clauses; see “[Bracket notation and assigner commands](#)”, [page 261](#) below.

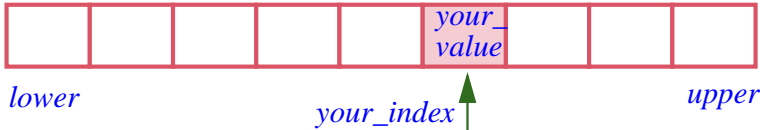
Note the precondition, requiring in both cases the index to be within bounds.

A typical use is, with *your_array*: *ARRAY [SOME_TYPE]* properly created as we've seen and *your_value*: *SOME_TYPE*:

← *valid_index*
appeared on page 259.

```
your_array.put (your_value, your_index)
```

which sets the corresponding array item:



*Updating an
array element*

overwriting any value previously entered there (including the default after initialization). Note the order of arguments: first the value to be written, then the index at which to write it.

After this call to *put*, the instruction

```
your_value := your_array.item (your_index)
```

will assign to *your_value* the item found at *your_index* in the array.

The postcondition of *put* as shown above expresses that, immediately after a *put*, the *item* value at the given index is the given value.

The examples for both *put* and *item* are only correct if the chosen *your_index* is within bounds. If this not guaranteed, then you should use

```
if your_array.valid_index (your_index) then  
    your_array.put (your_value, your_index)  
else  
    ...  
end
```

and similarly for *item*.

Bracket notation and assigner commands

For class *ARRAY* and others in this chapter the following notations, using brackets, are available:

```
your_value := your_array [your_index]  
    -- An abbreviation for your_value := your_array.item (your_index)  
your_array [your_index] := your_value  
    -- An abbreviation for your_array.put (your_value, your_index)
```

This is convenient for example in instructions such as

```
a [i] := a [i] + 1 [9]
```

more readable than *a.put (a.item (i) + 1, i)*. In such cases including several occurrences of array elements, especially with mathematical operations involved, `<ww-redonly>`[9<ww-redonly>] is better and follows mathematical practice.

There's nothing magical about the bracket notation, and it's not specific to arrays. To make it available for any class for which it makes sense, include an **alias** "`[]`" name for the corresponding feature, as with *item* in class *ARRAY*:

```
item alias "[]": G assign put is
  -- Entry at index i, if in index interval
require
  valid_key: valid_index (i)
do
  ... Implementation of the feature ...
end
```

Adding **alias** "`[]`" to the feature name indicates that the brackets are an “alias” for the feature name: another way to call it. As a result the notation

```
your_array [i]
```

is simply a synonym (an alias) for

```
your_array.item (i)
```

The declaration of *item* also specifies **assign put**. You may use such a clause for any *query*, listing an associated *command*. Its effect is to make the following assignment notation valid:

```
your_array .item (i) := your_value
```

merely as an abbreviation for a call

```
your_array .put (your_value, i)
```

to the command *put* which the **assign** clause has associated with *item*; such a command is called an *assigner command*. Because *item* now has both a

bracket alias and an assigner command, it is also legitimate to use form `<ww-redonly>[9<ww-redonly>]` as another synonym for the last call

```
your_array [i] := your_value
```

which achieves full reconciliation with traditional mathematical notation for arrays, vectors etc., while using the semantics of object-oriented operations.

A language note: most programming languages, from Pascal, C and C++ to Java and C#, offer such bracket notation for arrays, for both access (`your_array [i]`) and modification (`your_array [i] := your_value`). In these languages, however, the notation is specific to arrays, and arrays themselves are a special built-in notion. Eiffel has a different approach, treating `ARRAY` as a normal class with features `item` and `put`, for consistency with other data structures and the object-oriented approach (allowing, for example, a class to inherit from `ARRAY`). The language offers bracket notation as a synonym, through the `alias "[]"` construct. This construct is completely general and not limited to arrays: it's available in other structures studied later in this chapter, such as hash tables and linked lists, and you can apply it to any class that you write. In contrast, bracket notation in other programming languages is usually built-in and limited to arrays.

Resizing an array

Arrays have at any point in time a fixed `lower` and `upper` bounds, and hence a fixed number (`count`) of items. The precondition `valid_index` of `put` and `item` reflects this property. In most programming languages these properties are set once and for all, either statically (using constants bounds) or on creation. In Eiffel you can actually resize an array through `resize`:

```
resize (min_index, max_index: INTEGER)
  -- Rearrange array to accommodate indices down to min_index
  -- and up to max_index. Preserve existing items.
require
  good_indices: min_index <= max_index
ensure
  no_low_lost: lower = min_index or lower = old lower
  no_high_lost: upper = max_index or upper = old upper
```

Resizing is often indirect, through the procedure `force`. To change the value of an item, the default mechanism is `put (v, i)` with the precondition we've `seen`: `valid_index (i)`. This is usually the right approach; but it assumes that you know in advance how many items you'll need. If you have misestimated, and

the execution finds out after filling in all the entries of a 10-million-item array that it needs just one more, fixed-size arrays fixed-size arrays don't leave you any way out. Using *force* works in such cases:

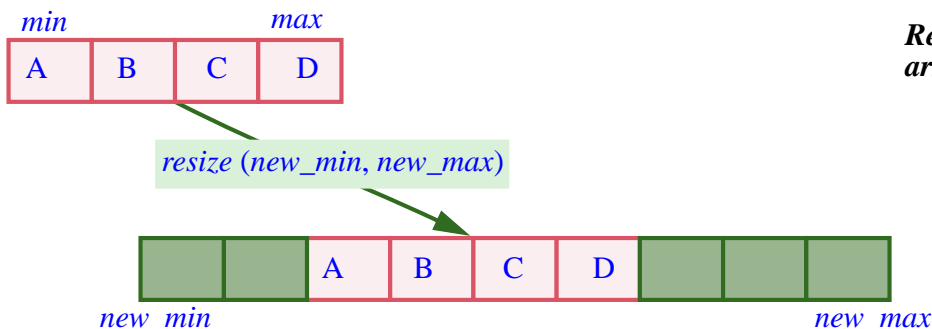
```

force (v: like item; i: INTEGER)
    -- Replace i-th entry, if in index interval, by v.
    -- Always applicable: resize the array if i falls out of current
    -- bounds; preserve existing items
ensure
    inserted: item (i) = v
    higher_count: count >= old count

```

Unlike *put*, procedure *force* has no precondition and so is always applicable. If *i* falls outside of the interval lower..*upper*, it will call *resize* to accommodate the requested entry.

Because of the continuous-memory implementation of arrays, resizing usually requires reallocating the array to a new place in memory and copying the old values:



Reallocation and copying are expensive, $O(\text{count})$ operations. As a result, *force* itself can be $O(\text{count})$, to be compared for the very fast, $O(1)$ cost of a standard *put*. Obviously, you should use *force* with care. Note that its implementation is prudent: if it has to call *resize*, it will make sure that the new size is sufficiently bigger than the previous one, so that for example a call

```

your_array.force (some_value, your_array.count + 1)

```

increases the size by more than one; the default policy is a 50% increase. So if you repeatedly use *force* in this style to extend an array at either end, only a few of the *force* operations will cause a *resize*.

Using arrays

An array of type *ARRAY* [*G*] represents a total function from the integer interval *lower..upper* to *G*. If after creation the bounds *lower* and *upper* don't change, or change only rarely, the implementation is highly efficient, since every access to the function, or modification of the function's value for a certain index in the interval, is $\mathbf{O}(1)$ and very fast. Arrays are then appropriate in situations where:

- You need to handle a set of values associated with an integer interval.
- Possibly after an initial period of allocating the array and filling up its initial values, the dominant operations are limited to index-based access and modification.

Because of the high cost of reallocation, arrays are not appropriate for highly dynamic data structures where elements come and go. In particular, it's expensive ($\mathbf{O}(\textit{count})$) to insert or delete an item if this implies renumbering the indexes, and hence shifting all the elements to the right (or left) of the insertion or deletion position. For structures with such behavior, you should use other structures studied later in this chapter.

Here is a summary of the cost of array operations.

Operation	Features in class <i>ARRAY</i>	Complexity	Comments
Index-based access	<i>item alias</i> " <code>[]</code> "	$\mathbf{O}(1)$	
Index-based replacement	<i>put alias</i> " <code>[]</code> "	$\mathbf{O}(1)$	
Index-based replacement outside of current bounds	<i>force</i>	$\mathbf{O}(\textit{count})$	Requires reallocating the array. (Only a fraction of successive <i>force</i> operations, will, however, cause such reallocation.)
New item insertion		$\mathbf{O}(\textit{count})$	Shifting all indices; not a common operation.
Removal		Not applicable	Can be done in $\mathbf{O}(\textit{count})$ by shifting indices; not a common operation.

10.5 TUPLES

Arrays are homogeneous: in an instance of `ARRAY [T]`, all items are of type `T`, or a type compatible with `T`. **Tuples** are similar to arrays, but may hold values of several specified types rather than just one. If you declare

```
tup: TUPLE [number: INTEGER, street: STRING, resident: PERSON]
```

the possible values for `tup` at run time are sequences of three or more components of which the first is of type `INTEGER`, the second of type `STRING` and the third of type `PERSON`, assumed to be an existing class. Such tuples could be useful, for example, in a census application, each of them recording the observation that at a certain `number` in a certain `street` lives a certain `resident`.

To denote a tuple value it suffices to write the successive components in brackets with commas in-between, yielding an expression, or *manifest tuple*, which can be used as argument to a routine call, or assigned to a tuple variable such as `tup`:

```
tup := [99, "Rue de Rivoli", Louvre_museum_curator]
```

[1]

With thanks to www.fun-trivia.com/playquiz/quiz113740d078a8.html.

The term “tuple” comes from mathematics: after the *pair* (two values, whose order matter) and the *triple* there’s the *quadruple*, the *quintuple* and (unless the term gets filtered out by parental controls) the *sextuple*, so it was natural for mathematicians to start talking about “*n*-tuples” for any *n*, denoting ordered sequences of *n* values.

Tuple types are not particularly exciting as a data structure — the way arrays, lists, hash tables, binary search trees and others each bring an original way to store and retrieve data, with its own efficiency advantages and limitations. In fact a straightforward implementation of tuples is through arrays. (Ignoring the specific type information we may look at a tuple as an `ARRAY [ANY]` where `ANY`, to be studied in more detail in later chapters, is the general high-level type covering all possible types.) Then for the complexity of tuple operations we can just use what we just found for arrays:

Operation	Tuple notation	Complexity	Comments
Component access	<code>t.comp</code>	$O(1)$	See below about the notations
Component replacement	<code>t.comp := value</code>	$O(1)$	
Removal		Not applicable	

The interest of tuple types lies elsewhere: as a language mechanism that enables you to describe simple structures in an clear and simple way, without resorting to classes. As part of this mechanism, we have already seen manifest tuples, as in `<ww-redonly>[1<ww-redonly>]` above, where the tags — `number`, `street`, `resident` — did not play any role. The tags are useful to access and set individual components of an existing (non-void) tuple; for example

after **[T1]** *tup.number* will have the value *99*. For setting component values, just note that the tags are treated exactly as attributes with associated “assigner commands”, enabling you to write, for example

```
tup.resident := some_person
```

← “*Bracket notation and assigner commands*”, page 261.

All this suggests that we could do without tuple types by using *classes* such as

```
class CENSUS_RECORD feature
  number: INTEGER assign set_number
  street: STRING assign set_street
  resident: PERSON assign set_resident
  set_number (n: INTEGER) do number := n ensure number = n end
  ... set_street, set_resident like set_number ...
end
```

which gives us exactly the same operations on *cr* of type *CENSUS_RECORD* as on *tup* above: to access components, *cr.number* and such; to set components, *cr.resident := some_person* and such. Tuples are useful when you need composite values (sequences of components of known types) and no other operations than component accessing and setting, with *no* preconditions; they save the need for writing simple classes such as *CENSUS_RECORD*. For that reason, tuples are also called *anonymous classes*. As soon as you need anything more sophisticated, they won’t do any more: you should declare a class — not anonymous — modeling your exact needs.

To finish with the language mechanism, note that tags do not affect the tuple’s type; in fact they are optional. So you can also write the above type as *TUPLE [a: INTEGER, b: STRING, x: PERSON]*, or just *TUPLE [INTEGER, STRING, PERSON]* if you don’t need to access or set the components by name.

Syntactically, such tuple types look like generically derived class types, like *LIST [T]*; indeed the concepts are similar, but there is no class *TUPLE* because it would have to admit an arbitrary number of parameters, whereas a generic class always takes a fixed number (one parameter in *ARRAY [G]* and *LIST [G]*, two in *HASH_TABLE [G, KEY]* studied below). With tuple types you can describe sequences of any length: *TUPLE* with no parameters covers all sequences, *TUPLE [T]* sequences of at least one element with the first of type *T*, and so on.

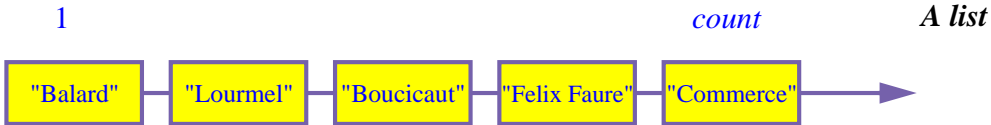
This observation also determines *conformance* properties: you may assign an expression of type *TUPLE [T, U, V]* to a variable of the same type, of type *TUPLE [T, U]*, of type *TUPLE [T]*, or just *TUPLE*. As noted, there is no class *TUPLE*, but *TUPLE* denotes a type, which covers all possible tuples, of any length and any component types.

We will find tuple types particularly useful in connection with *agents*, covering applications such as iteration and event-driven programming.

→ Chapter 20.

10.6 LISTS: LINKED, ARRAYED, MULTI-ARRAYED

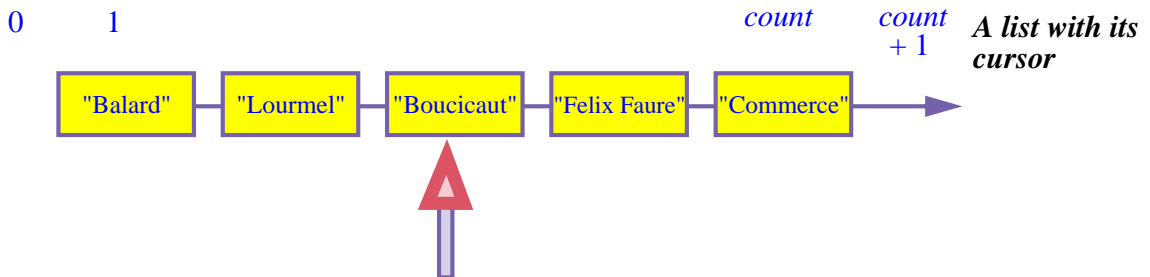
A *list*, also known as a *sequence*, is a container keeping elements in a certain order, usually the order of insertion. Mathematically, it represents a total function from the interval $1..count$ to G ; this seems similar to arrays, but the big difference is that *count* can vary freely, as you insert new elements.



The figure illustrates a list, made of five items. The arrow is there simply to highlight that order matters. The same elements organized in a different order would make up a different list.

As for arrays and other structures where elements are numbered, we systematically start the numbering at 1. Probably because it's convenient for the manipulation of addresses in *hardware* memory, where the units are powers of two, to count from 0 to 2^n-1 for some n , C and some other programming languages force the numbering of items in arrays and other structures to start from 0, so that a structure of *count* elements will be indexed from 0 to *count*-1. This means that you constantly have to remember to add or remove one. It seems much simpler to start from 1 and follow everyone's understanding — going back to early childhood — that the i -th element has index i . On your hand the thumb is the first finger (not the zeroth), and the middle finger is the third, not the second.

The classes representing lists in EiffelBase — *LIST*, *LINKED_LIST*, *ARRAYED_LIST*, *MULTI_ARRAYED_LIST* and a few others — treat a list not just as a collection of elements but as a *machine* which at any point in its existence has a *state* characterized by a *cursor*:



This notion is not new; when we manipulated a metro line as a list of stations we already had a cursor.

Having a cursor makes it convenient to perform the basic list operations — accessing, inserting or deleting an item — by ensuring that the corresponding routines have a simple specification: instead of asking you to specify a list position, they work relatively to the cursor position, as in “delete the element at cursor position”.

← [“Animating a metro line”, page 164.](#)

If in the current state of a list the cursor is either “before” or “after” we say that it is “off”:

```

off: BOOLEAN
    -- Is there no current item?
ensure
    definition: Result = (is_after or is_before)

```

Further invariant clauses express the properties of these queries (postconditions are also possible):

```

before_definition: is_before = (index = 0)
after_definition: is_after = (index = count + 1)
off_definition: off = (index = 0 or index = count + 1)

```

Other queries on the cursor position include:

```

is_first: BOOLEAN
    -- Is cursor on first item?
ensure
    valid_position: Result implies (not is_empty)
is_last: BOOLEAN
    -- Is cursor on last item?
ensure
    valid_position: Result implies (not is_empty)

```

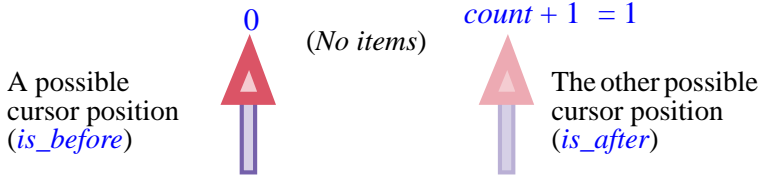
themselves relying on

```

is_empty
    -- Are there no items?

```

A list can indeed be empty, in which case *is_first* and *is_last* always yield false as implied by the relevant invariant clauses: the cursor can only be on the first item if there is at least one item. Do not forget the Extreme Cases Principle: it is essential to make sure that our conventions still work well in such border cases. For an empty list the previous figure becomes



An empty list and its two possible cursor positions

with no item. In this case *count* is zero, so the maximum *index* position satisfying the invariant, $count + 1$, is one. In such an empty list the cursor can be in either position 0 or position 1. In either case *off* will hold, hence the invariant clause

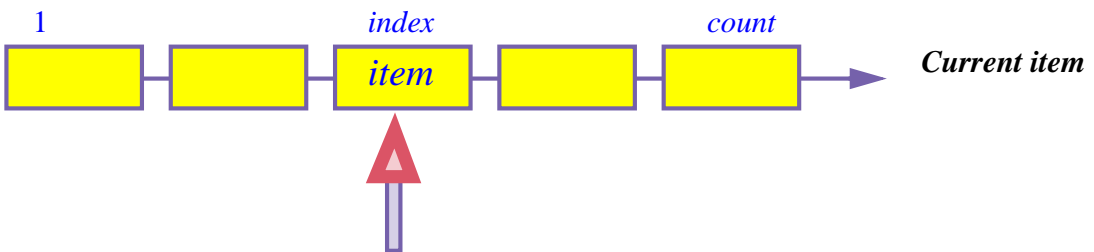
empty_constraint: *is_empty* **implies** *off*

Note the repeated accumulation of invariant clauses to express, little by little, what we understand of our own object structures:

Touch of Methodology: Using invariants

Use invariant clauses to make explicit the consistency properties of the classes you design, and to check (in particular by considering extreme cases, in line with the Extreme Cases Principle) that these properties are sound and compatible.

To access the list item at cursor position:



you will use

```

item: G
  -- Item at cursor position
  require
    not_off: not off

```

This query returns a result of type *G*, the generic parameter of the list classes (*LIST [G]*, *LINKED_LIST [G]* etc.). Note the precondition: in an *off* state — including for an empty list — there is no “current item”.

Cursor movement

You have a number of commands at your disposal to move the cursor around. The following will bring the cursor to the beginning or end of a list:

```

start
  -- Move cursor to first position (no effect if empty)
ensure
  at_first: (not is_empty) implies is_first

finish
  -- Move cursor to last position (no effect if empty)
ensure
  at_last: (not is_empty) implies is_last

```

A call to *start* ensures *is_first*, and a call to *finish* ensures *is_last*, but only, for reasons just discussed, for non-empty list. The postconditions express this.

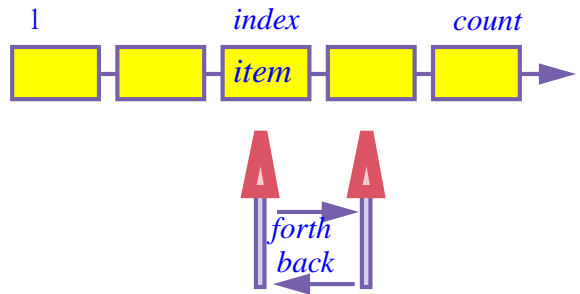
You can also move the cursor by one position:

```

forth
  -- Move cursor to next position
require
  not_after: not is_after
ensure
  moved_forth: index = old index + 1

back
  -- Move cursor to previous position
require
  not_before: not is_before
ensure
  moved_back: index = old index - 1

```



Note the preconditions, guaranteeing that the index remains within bounds as specified by the earlier invariant clauses *non_negative_index* and *index_small_enough*. You may also move the cursor to a specified position: ← Page 269.

```

go_i_th (i: INTEGER)
  -- Move cursor to next position
require
  valid_cursor_position: i >= 0 and i <= count + 1
ensure
  position_expected: index = i

```

Iterating on a list

One of the most common manipulations on a list is to apply a given operation to every item in turn. Let's assume this operation is given by a routine

```
your_operation (x: G)
```

In dealing with metro lines and their stations we've already seen the scheme for applying *your_operation* to every list item; the general form is the loop

```
from
    your_list.start
until
    your_list.after
variant
    your_list.count - your_list.index + 1
loop
    your_operation (your_list.item)
    your_list.forth
end
```

This is for applying an operation to some existing list *your_list* from your program. The scheme also appears within the list classes themselves to perform traversals of the *current* list, using unqualified calls *start*, *after*, *forth* without “*your_list.*” We'll see an example shortly with the routines *search* and *has* that search for a value among the items of a list.

There are other forms, for example to apply a certain operation to all elements of a list *up to* and excluding the first that satisfies a certain condition:

```
from
    your_list.start
until
    your_list.after or else your_condition (your_list.item)
variant
    your_list.count - your_list.index + 1
loop
    your_operation (your_list.item)
    your_list.forth
end
```

Such a scheme is an example of an *iterator*:

Definition: Iterator

An **iterator** on an object structure is a mechanism for applying any given operation to all items of the structure, or to all items satisfying any given condition.

Isolating general schemes is good; making them reusable, so that one doesn't have to code them anew each time, is better. The basic iterator mechanisms are indeed captured by features of class *LIST* such as *do_all* and *do_if*, which capture the preceding loop structures once and for all, so that you may write the last two examples as

```
your_list.do_all (agent your_operation)
your_list.do_if (agent your_operation, agent your_condition)
```

where **agent *your_operation*** denotes an object that represents the procedure *your_operation* ready to be applied to each item, and **agent *your_condition*** similarly represents the query. To understand the details we'll need to study first the general notion of agent in a later chapter. Even with agents at your disposal, you'll probably have opportunities to write explicit list traversals (*iterations*), using the above schemes as your guides.

→ Chapter 20.

An example of implementation using an iterator mechanism, shared by all the list classes, is the procedure *search* for finding an element in a list. Its text looks like this:

```
search (v: G) is
  -- Move cursor to first position, at or after current position,
  -- where item value is v; if none go to is_after position.
do
  from
    if before and not is_empty then
      forth
    end
  until
    is_after or else item = v
  loop
    forth
  end
end
```

This version compares *v* and *item* through basic equality =; it's also possible to use object equality, ~.

This feature is a command, which will bring the cursor to the next position, if any, where the list includes the sought value, and to the extreme right if there's none. This lets you use *search* repeatedly to search for successive occurrences of a value. The procedure is also used in the implementation of *has*, the query to tell you whether a value appears at all:

```

has (v: G) is
  -- Does structure include an occurrence of v?
  local
    original_index: INTEGER
  do
    original_index := index
    start
    search (v)
    Result := not is_after
    go_i_th (original_index)
  end

```

As a query, *has* should leave the object structure in the state where it found it; it uses a local variable *original_index* to record the initial cursor position and return to it, through *go_i_th*, at the end.

Both *search* and *has* require $O(\text{count})$ time (maximum and average).

Adding and removing elements

To add an item to a list — at the beginning, the cursor position, or the end — you may use one of the operations with the following specifications:

```

put_front (v: G)
  -- Add v to beginning; do not move cursor
put_left (v: G)
  -- Add v to left of cursor position; do not move cursor
  require
    not_before: not before
put_right (v: G)
  -- Add v to right of cursor position; do not move cursor
  require
    not_after: not after
extend (v: G)
  -- Add v to end; do not move cursor

```

Unlike the last two examples, which showed routine implementations, these are just interface specifications of the corresponding EiffelBase features.

As the comments indicate, these procedures are designed to have no effect of the cursor, since there is no reason an insertion should change the currently active position in the list.

In many cases the implementation does change the cursor temporarily; for example it's possible to implement *extend* (*v*) as

```

original_index := index
finish
put_right (v)
go_i_th (original_index)

```

with an integer variable *original_index*, as in *has*, to record the initial index position, enabling the command to restore the cursor position at the end.

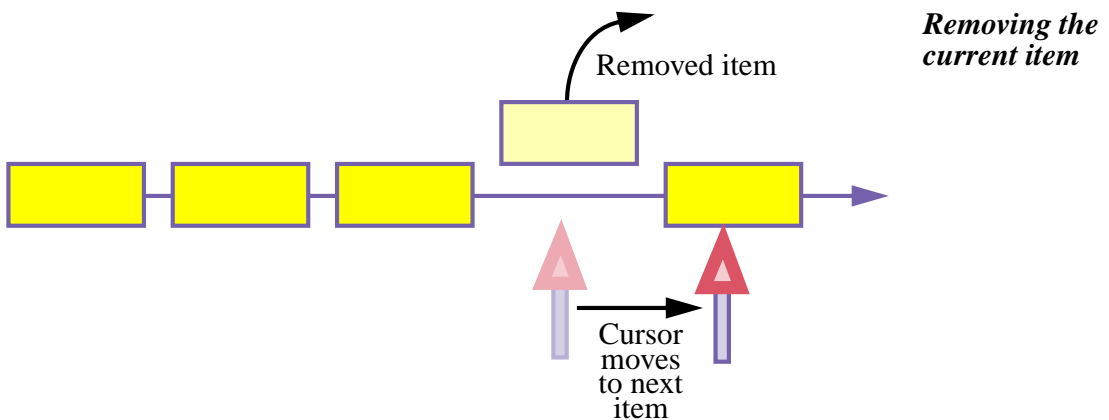
To delete elements you may use

```

remove
  -- Remove item at cursor position; move cursor to right neighbor
  -- (or to is_after if no right neighbor).
require
  item_exists: not off
ensure
  removed: count = old count - 1
  after_when_empty: is_empty implies after

```

In this case the cursor has to be moved because the item to which it was pointing goes away:



There's also *remove_left* and *remove_right*, acting on positions next to the cursor, which do not change the cursor position. Write their specifications (signature, header comment, contract) as an exercise.

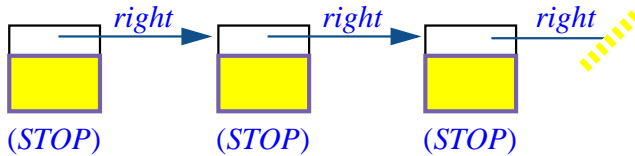
Linked lists

We've now seen the basic properties and features of lists independent of any implementation and turn our attention to specific implementations. As in the rest of this chapter the implementations will not be described in detail, but we'll see the principal ideas. You can read the class texts from EiffelBase for the full picture.

These classes are all “descendants” of the class *LIST* in the sense of *inheritance*, to be studied soon. This means in particular that they don't repeat common elements, but move them to the class at the highest level of generality in each case.

→ Chapter 18.

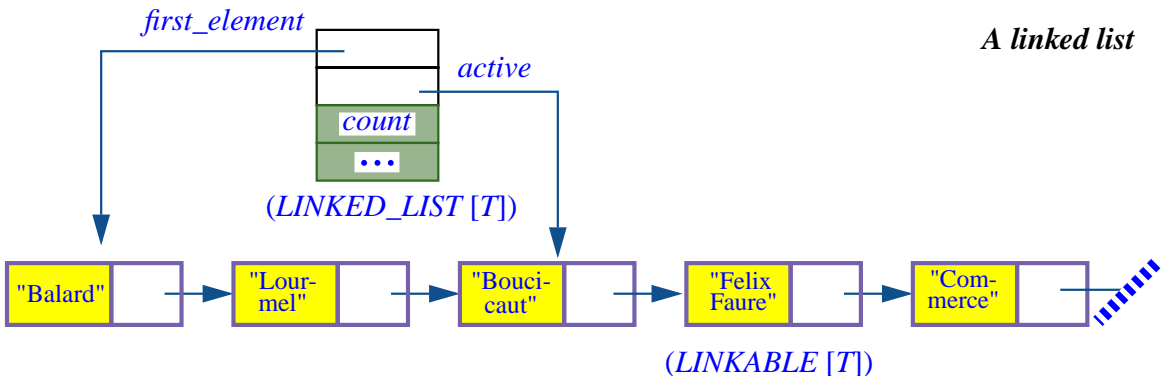
The first implementation is in class *LINKED_LIST*. In our work with metro stations we saw the technique of *linking* elements of a sequential structure:



Linking stations

← This figure first appeared on page 118.

We can generalize this — thanks to the genericity mechanism — to arbitrary structures. An instance of *LINKED_LIST [T]* for some type *T* will refer to zero or more linked cells, or “linkables”, each containing a value of type *T* and a reference to a possible other such linkable:

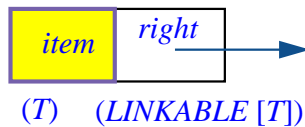


A linked list

As the figure indicates, the implementation involves two classes:

- The bottom object is an instance of *LINKED_LIST [T]*. It provides access to the list as a whole, but doesn't directly contain any of the list items; its fields denote general information about the list, such as *count*, the number of elements, if implemented by an attribute (it could also be a function), and references to list cells such as *first_element* indicating the first cell and *active* indicating the element at cursor position. Such an object is known as a **list header**.
- The other objects represent list cells; they are instances of a class *LINKABLE*, also generic and using the same actual generic parameter, here *LINKABLE [T]*.

Class *LINKABLE* serves implementation purposes; in normal usage client applications that need linked lists will only see *LINKED_LIST*. *LINKABLE* represents a very simple notion of list cell that can be linked to other similar cells; a typical instance looks like this:



*An instance of
LINKABLE [T]*

The implementation of *LINKED_LIST* routines relies on features from *LINKABLE*: the queries

```

item: G
    -- Value in cell
right: LINKABLE [G]
    -- Next item

```

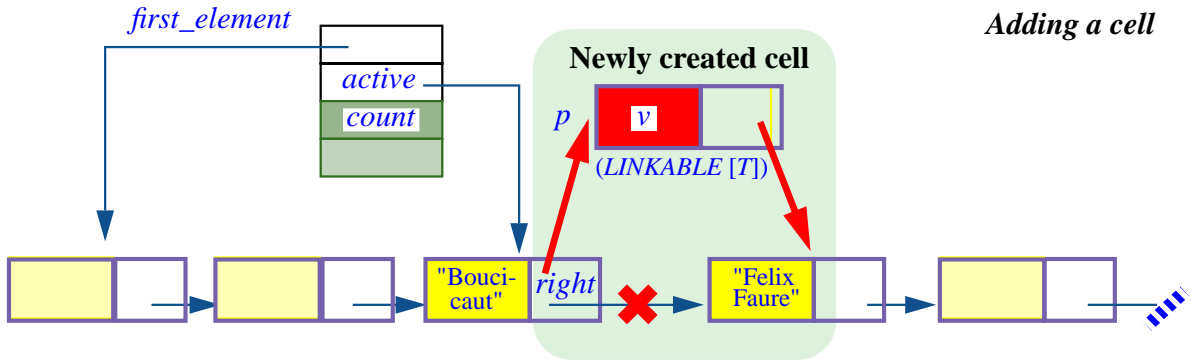
and the associated setter commands:

```

put (x): G
    -- Set item's value to x.
    ensure
        set: item = x
put_right (other: LINKABLE [G])
    -- Chain to other.
    ensure
        set: right = other

```


Here for example is a picture of how class *LINKED_LIST* implements the command *put_right*, which as specified earlier must add an element to the right of the cursor without moving the cursor. For a linked list, it suffices to create a new *LINKABLE* cell and update the chaining: ← Page 275.



In the implementation of the routine, the illustrated operation uses two calls to *put_right* from *LINKABLE*, highlighted below. Note how it must also include special treatment to handle the *is_before* case properly:

```

put_right (v: G)
  -- Add v to right of cursor position; do not move cursor
  require
    not_after: not after
  local
    p: LINKABLE [G]                -- The cell to be created
  do
    create p.make (v)
    if is_before then              -- Special is_before case:
      p.put_right (first_element)
      first_element := p
      active := p
    else                             -- The most common case:
      p.put_right (active.right)
      active.put_right (p)
    end
  ensure
    next_exists: active.right /= Void
    inserted: (not old is_before) implies active.right.item = v
    inserted_before: (old before) implies active.item = v
  end

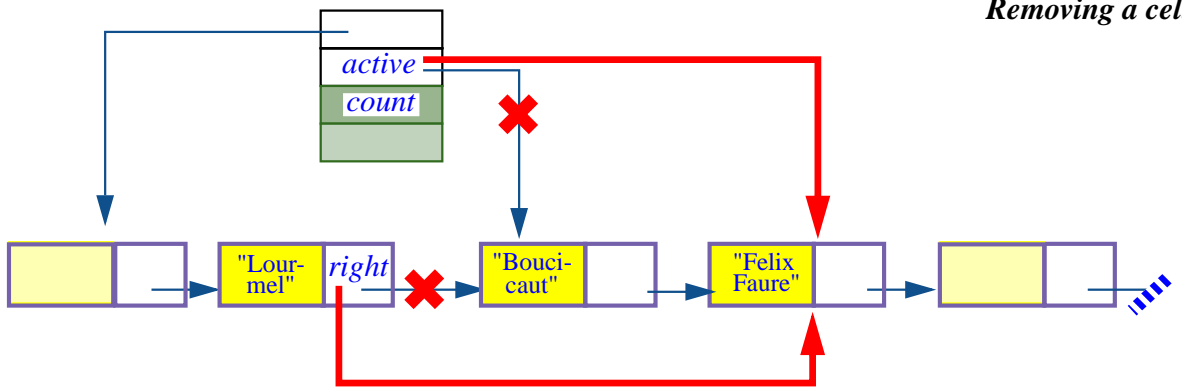
```

This routine is from *LINKED_LIST*; it includes three calls to a routine with the same name from *LINKABLE*. The reuse of the name is part of the notational convention ("Standardizing feature names for basic operations", page 252) and causes no ambiguity since *p* is of type *LINKABLE*.

As most object structure manipulations involving some juggling of references, this algorithm requires care to ensure that it works correctly in all details. The difficulty is even more apparent in the next example, item removal. Programming with references is indeed a delicate area; we'll draw below some methodological consequences of this observation.

→ "[Programming with references](#)", page 282.

The procedure *remove* discards the element at cursor position; as we've seen, the specification states that the cursor will move to the position immediately to the right: ← Page 276.



The implementation must, as illustrated, change two references:

- It must reattach the *right* link of the element just before the cursor position (with item value "Lourmel" at the bottom of the figure) to bypass the element at cursor position.
- To update the cursor as required, it must reattach the *active* link of the *LINKED_LIST* object to the element (here "Felix Faure") just after the previous cursor position.

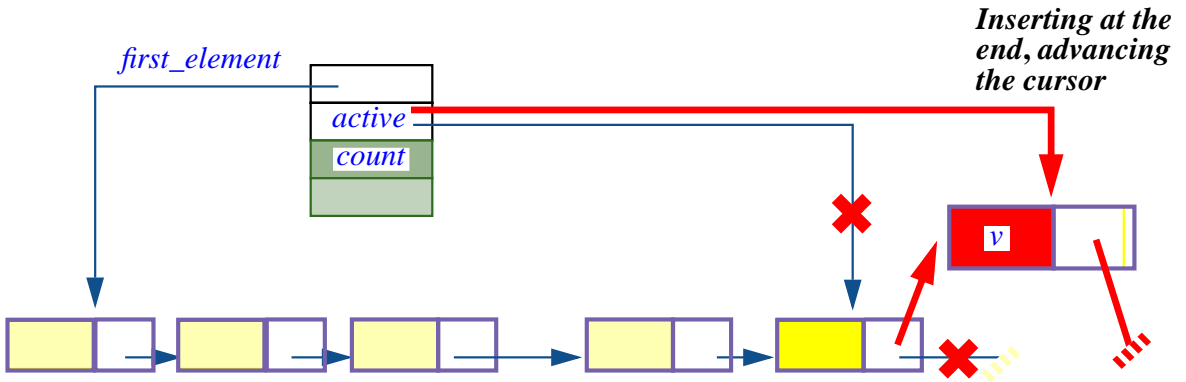
Here too you may look up the actual implementation — routine *remove* in *LINKED_LIST* — but the details are more intricate than for *put_right*, as there are several special cases, including when the cursor is on the first or last item. It may help to read first the text of routine *remove_right*, somewhat simpler.

These examples give a good idea of what's involved in implementing the list operations the *LINKED_LIST* way. As to performance:

- The complexity is $O(1)$ for operations that need only to perform operations at or around the cursor position: *put_left*, *put_right* and the removal operations. (Obtaining $O(1)$ for some of these may require more attributes in *LINKED_LIST*, for example a reference *previous* to the cell left of cursor.) Basic cursor movement *start* and *forth* are also $O(1)$.

- Operation that may need to traverse the list are $O(count)$. This is the case, as we've already seen independently of the choice of implementation, for *search* and *has*, and also applies here to the general cursor movement, *go_i_th*, as well as to *finish* (implemented as *go_i_th(count)* and *back* (implemented as a *start* followed by *go_i_th(count - 1)*)

An interesting case is *extend*, to add an element at the end of the list. As noted, ← Page 276. this can be implemented as *finish* followed by *put_right*, and a *go_i_th* if we need to restore the cursor position. All three operations are $O(count)$. But often you will need, for example when initializing a list, to add items repeatedly at the end. If indeed you can let the cursor remain on the last element (query *is_last*), then *extend* will just perform a *put_right* followed by a *forth*, and hence will be $O(1)$:



Here is the complexity summary, as for other structures in this chapter. First the cursor-position operations:

Operation	Features in class <i>LINKED_LIST</i>	Complexity	Comments
Insert at cursor position	<i>put_right</i> , <i>put_left</i>	$O(1)$	For operations left of cursor, obtaining $O(1)$ may require a <i>previous</i> attribute.
Remove at cursor position	<i>remove</i> , <i>remove_right</i> , <i>remove_left</i>	$O(1)$	
Insertion at end, if cursor already there	<i>extend</i>	$O(count)$	

Then cursor movements:

Move cursor to first	<i>start</i>	$O(1)$	For operations left of cursor, $O(1)$ may require a <i>previous</i> attribute.
Move cursor to last	<i>finish</i>	$O(\text{count})$	For operations left of cursor, $O(1)$ may require a <i>previous</i> attribute.
Move cursor one step right	<i>forth</i>	$O(1)$	For operations left of cursor, $O(1)$ may require a <i>previous</i> attribute.
Move cursor one step left	<i>back</i>	$O(\text{count})$	For <i>put_left</i> , relies on <i>previous</i> attribute in class <i>LINKED_LIST</i> .

Finally, global operations that may require a traversal:

Insert at end, if cursor not there	<i>extend</i>	$O(\text{count})$	
Search	<i>search, has</i>	$O(\text{count})$	

Programming with references

This routine provides a good illustration of techniques of *programming with references*: reattaching references around to perform insertions into linked object structures and (the next example) deletions from these structures. Such manipulations can be quite delicate, as even a basic routine such as *put_right* already illustrates. You must take into account all possible cases including empty or almost-empty structures or those in which the cursor (if any) is in a special case, and pay special attention to the ever-present possibility of void references, as they may not be targets of feature calls.

Such matters should not be the stuff of application programs:

Touch of Methodology: **Reference Programming Principle**

Non-trivial manipulations of references, typically, for inserting and removing items to and from object structures, should appear not in the application-oriented parts of programs but in library classes expressly devised to implement such structures.

The “*application-oriented parts of a program*” are those dealing directly with the program’s intent: processing calls (in the software running your cell phone), selling securities, typesetting text... This seldom directly involves tricky reference manipulations of the kind just seen, although they are often useful for the *implementation* of application-oriented concepts. Your text-processing system may use a linked list of paragraphs, but juggling with the references to enter a new paragraph into the list is not a text-processing issue, it’s a list issue, and should be handled in software components that deal with object structures in general.

If you have to implement such manipulations yourself, the Reference Programming Principle directs you to separate them from the application proper, putting them into special “supporting technology” clusters.

Fortunately, you often won’t have to write such support software yourself. Modern development environments provide libraries of components dealing with the basic kinds of object structures; EiffelBase, covering all the structures described in this chapter and many others, is an example, developed by many people over many years. For the common case of

A consequence of the above advice is an injunction to use such libraries:

Touch of Methodology:
Fundamental Data Structure Library Principle

For fundamental data structures and algorithms, use components from a basic library such as EiffelBase, if applicable, rather than developing your own implementations.

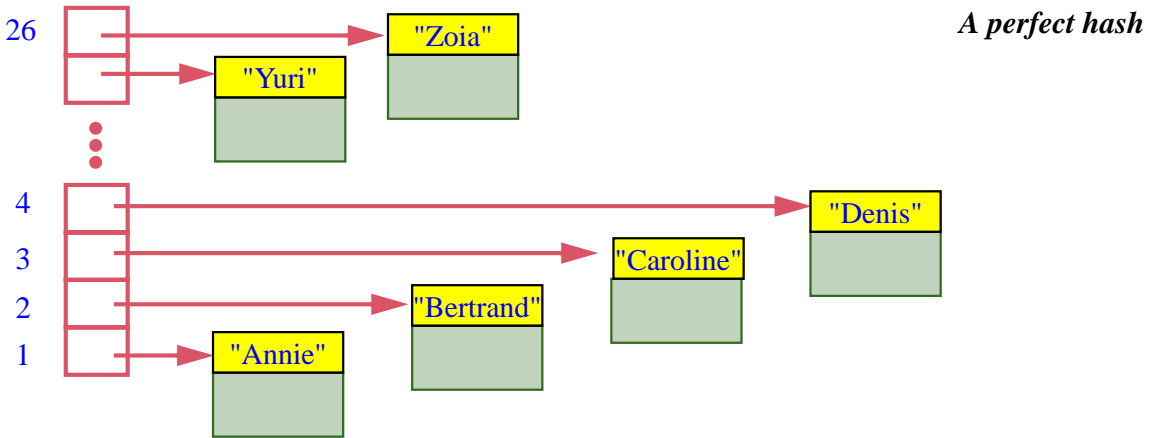
Class *LINKABLE [G]* has queries *item* and *right* and the associated setter commands *put* and *put_right*, used by the implementation of *LINKED_LIST* routines. These

10.7 HASH TABLES

Arrays represent structures indexed by integers. What if we want some key other than integers? Strings are a common example. You may need containers where the access criterion is a character string, such as:

- A directory of people — a container where each object represents information about one person; you’ll want to retrieve these objects through people’s names, in the same way that you find a person, in a paper directory, by looking up his name.
- A collection of Web pages, as maintained by a search engine: it’s indexed by all the words that appear on the pages.

Assume for a moment that in the first example all the people in the directory have names starting with a different letter: **A**nnie, **B**ertrand, **C**aroline, ... Then you could use an array of twenty-six entries, each corresponding to a letter code, 1 for **A**, 2 for **B** and so on:



We have **hashed** the keys (the strings representing the names) into integer values in the interval $1..26$. “Hashing” is taken here in the sense of mincing up food into small pieces. Specifically:

Definition: Hash function

A **hash function**, for a set K of possible keys, is a function h that maps K into some integer interval $a..b$.

In other words, for any $key \in K$, the function gives you a value $i = h(key)$ such that $a \leq i \leq b$.

In practice the interval is usually of the form $0..capacity-1$ for some integer *capacity*, with $h(key)$ of the form $f(key) \bmod capacity$ for some basic function f returning an integer. The array will then be of size *capacity*.

The example uses a very primitive hash function that simply returns, for a string key, the integer code of the first letter, in the interval $1..26$. A slightly more sophisticated hash function would take the ASCII codes of *all* characters in the string and add them, then take the remainder by *capacity*.

← ASCII is the standard encoding of basic characters, with values from 0 to 255.

The hash function depends only on the key of each item, not on the number of items *count*, so if *count* is the measure of our problem’s size its execution will be $O(1)$. (If we take into account the length l of keys, it may be $O(l)$, but we may assume that the hash function only uses the first K characters of the key for some constant K .)

The assumption behind the example was that each name started with a different letter, giving a different hash value. A hash function that gives a different value for every element of a given set of keys is called a **perfect hash** for those keys. With a perfect hash, insertion and search are $O(1)$.

In most cases, we won't be able to get a perfect hash, even with a better hash function such as the sum of all codes modulo *capacity*. A **collision** occurs — with a non-perfect hash function — when two different keys give the same hash value. A good hash function will cause fewer collisions (it's in this sense that we can say that the second example, sum modulo *capacity*, is generally "better" than the first), but won't usually avoid them completely. In fact, if the hash function computes its result modulo *capacity*, collisions are inevitable as soon as we deal with more than *capacity* keys. The implementation of hashing must be able to handle them.

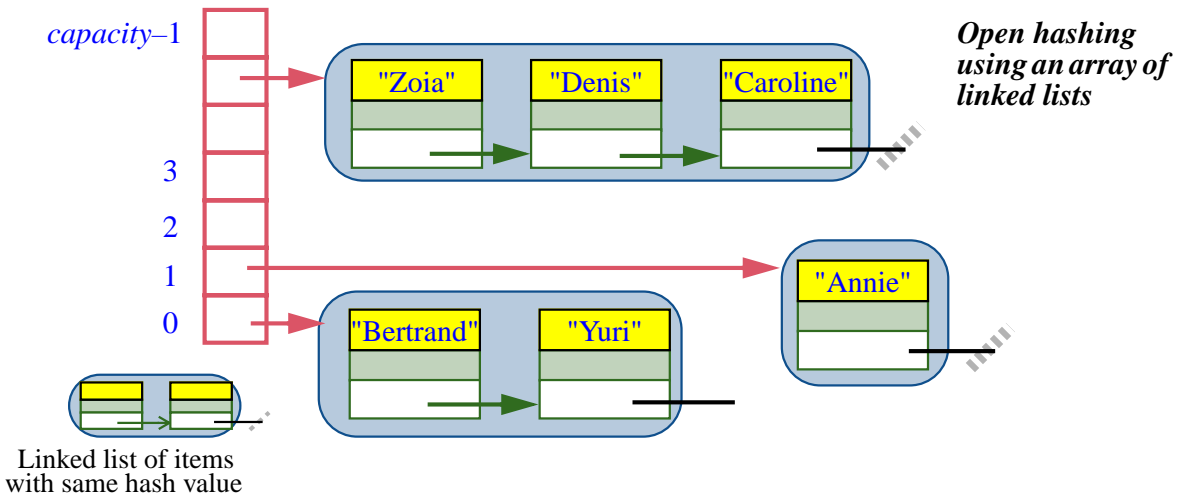
One technique is **open hashing**, which combines arrays with linked lists. In the last figure, with a perfect hash, the array directly contained items and would have been declared as

```
ARRAY [G]
```

but with open hashing we'll use an array of *linked lists* of objects:

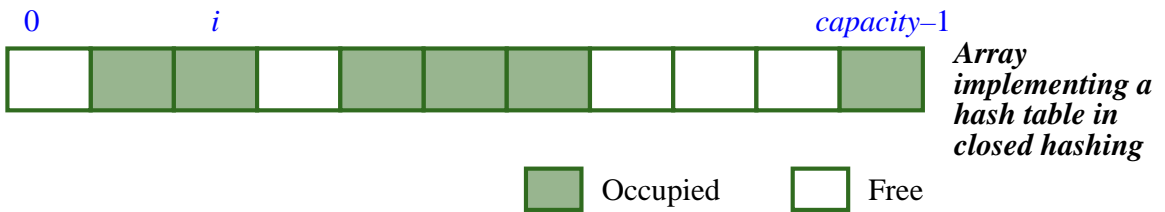
```
ARRAY [LINKED_LIST [G]]
```

In each entry of the array, for a certain index *i*, you find the list of objects whose keys hash to *i*:

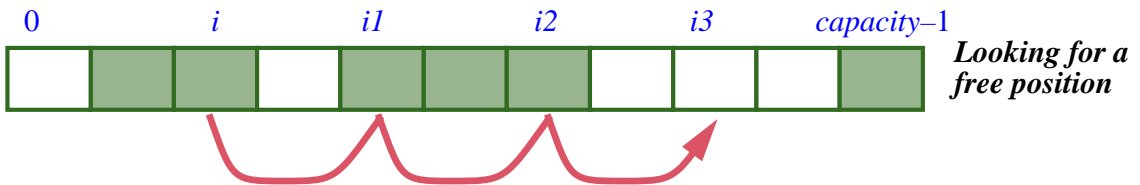


With open hashing we search for an item, or insert it, by first hashing its key into an array index, then performing a sequential search in the associated list. The cost is $O(1)$ for the first operation and $O(c)$ for the second, where c is the collision factor — the average number of keys hashing to a given value. If the array size *capacity* is constant, the value of c for a large number *count* of items and an evenly distributed hash function will be $O(\text{count} / \text{capacity})$, that is to say $O(\text{count})$. To avoid this linear behavior we would need periodically to resize the array; but then it's usually better to use the other technique for collision resolution, **closed hashing**.

With closed hashing — as applied by the EiffelBase class *HASH_TABLE*, which you may study for a deeper understanding of hashing — there is a single *ARRAY[G]*. At any time some of its positions will be occupied and some free:



If for an insertion the hash function yields an already occupied position, for example the one marked i above, the mechanism will try a succession of other positions — $i1, i2, i3$ below — until it finds a free one:



A common technique, if the hash function yields a first candidate position $i = f(\text{key}) \bmod \text{capacity}$, is to try successive positions $i + \text{increment}$, $i + 2 * \text{increment}$, $i + 3 * \text{increment}$ and so on, all modulo *capacity*, where *increment* (2 on the above figure) is $f(\text{key}) \bmod (\text{capacity} - 1)$. This is the algorithm used by *HASH_TABLE* in EiffelBase; see its implementation routine *search_for_insertion* if you want to study it in detail.

To guarantee that the process terminates (meaning that the corresponding loop has a variant) the algorithm must always find an empty slot. This is guaranteed by an appropriate choice of parameters and by a policy that reallocates the array — *resize* is essential here — if it fills up. In fact we mustn't wait until the last minute: reallocation should occur as soon as the fill factor reaches a preset limit, which class `HASH_TABLE` sets at 80% (see the feature `Max_occupation` in the class). What's truly amazing is that with this policy, and a good choice of hash function, search and insertion in a hash table are essentially $O(1)$. (For the theoretical complexity analysis leading to this property, see the references at the end of this chapter.)

This property means that for all practical purposes you may see hash tables as almost as good as arrays, generalized to arbitrary keys, not just integers, as long as the keys are “hashable”. Strings, for example, are hashable, so you may consider a hash table of string-identified objects as if it were an array indexed by strings rather than integers.

This is a remarkable result, since *really* indexing by strings would lead to impossibly huge structures. Consider for example strings of at most 7 lower-case letters; the number of possibilities is approximately 26^7 , or 8 billions, but it would be absurd to use an array of that size even if we had the memory, since any practical use needs only a small subset of these possible strings. By hashing the strings we allocate just a little more space than what we actually need (per `Max_occupation` noted above) and still get time behavior comparable to that of an array.

Finding hash functions that yield such efficient behavior is somewhat of an art; you can again take inspiration from the function used in class `HASH_TABLE`.

That class, `HASH_TABLE[G, KEY]`, is our first example with two generic parameters rather than just one; `G` represents the type of items and `KEY` the type of their keys. You may use it for example to declare a hash table of objects representing persons, indexed by their names, as

```
personnel_directory: HASH_TABLE [PERSON, STRING]
```

Here are some of the fundamental features of `HASH_TABLE`. There's a single creation procedure `make`; to create a hash table use for example

```
create personnel_directory.make (initial_size)
```

where `initial_size` is some positive integer. It doesn't matter much what value you select; as the name suggests, this is just a hint for the initial allocation. If you are too far below the real need, you'll just pay for one more resizing (automatic, of course) at run time

Next, the main queries. To find out if there's an item for a certain key use

```
has (k: KEY) BOOLEAN
```

To obtain the item associated with a given key, if any:

```
item (k: KEY) alias "[]": G assign put
  -- Item associated with k, if any; otherwise default value of type G
ensure
  default_value_if_not_present:
    not (has (k) implies (Result = computed_default_value)
```

The postcondition indicates that if there *isn't* an item for the given key, the result is the “default value” of type *G* (zero for numbers, false for booleans, void for references). This is not a good way to test for the presence of an item in a hash table, since there could be an item with the default value; so if you are not sure whether the key appears, use *has* first.

The *alias* "[]" specification indicates, as with *item* for arrays, that bracket notation is available for the item associated with a certain key: you may write

← “*Bracket notation and assigner commands*”, page 261.

```
personnel_directory ["Isabelle"]
```

as a synonym for

```
personnel_directory.item ("Isabelle")
```

The bracket form is shorter and we'll use it whenever applicable.

To insert an item into a hash table, you will need to provide both the item and its key, as in

```
personnel_directory.put (that_person, "Isabelle")
```

[10]

even if the key is in fact a property of the item, as in

```
personnel_directory.put (that_person, that_person.name)
```

The class offers four insertion operations with the same signature:

```
put (new: G; k: KEY)           -- This is the assigner command for item
force (new: G; k: KEY)
extend (new: G; k: KEY)
    require
        not_present: not has (k)
replace (new: G; k: KEY)
```

Among them, *extend* has a precondition stating that it's only applicable if the key is not already used; the other three are always applicable. A “**note**” clause at the beginning of the class explains when to use each variant; since it says exactly what there is to say I am reproducing it here, omitting some details:

Insertion variants for hash tables (from the text of class *HASH_TABLE*)

- Use *put* if you want to do an insertion only if there was no item with the given key, doing nothing otherwise.
- Use *force* if you always want to insert the item; if there was one for the given key it will be removed.
- Use *extend* if you are sure there is no item with the given key, enabling faster insertion.
- Use *replace* if you want to replace an already present item with the given key, and do nothing if there is none.

In the first two cases the procedure will set the value of the boolean query *found*, enabling you to find out, after insertion, if there already was an element with the given key.

Since *item*'s declaration named *put* as the associated assigner command:

```
item (k: KEY) alias "[ ]": G assign put
  -- Item associated with k, if any; otherwise default value of type G
ensure
  default_value_if_not_present:
    not (has (k) implies (Result = computed_default_value)
```

where *put* is declared as *put* **alias** "[]", bracket notation is available for calling this procedure, so that you may insert an element into a hash table through an assignment-like form:

```
personnel_directory ["Isabelle"] := that_person
```

which is just a shorthand for the explicit call to *put* on the previous page <ww-redonly>[10<ww-redonly>].

To remove an element with a given key, use

```
remove (k: KEY)
```

This has no effect if the key wasn't present; you can find out afterwards through the query *removed*.

Calling *clear_all* will remove all the current entries.

Throughout these operations you don't have to worry about the size of the data structure; thanks to the resizable nature of Eiffel arrays, the routines will take care of maintaining enough space for all the current items, plus some breathing space as required by the algorithms (fill factor of at most *Max_occupation*).

If you explicitly want to change the size, a call to *accommodate* (*n*: *INTEGER*) will ensure that the table can accommodate *n* items; it won't discard any existing one.

Here is a summary of the cost of hash table operations.

Operation	Feature in class <i>HASH_TABLE</i>	Complexity	Comments
Key-based access	<i>item</i> , <i>has</i>	O (1)	
Key-based insertion	<i>put</i> , <i>force</i> , <i>extend</i>	O (<i>count</i>)	
Key-based replacement	<i>replace</i>	O (1)	
Removal	<i>remove</i>	O (1)	

As you start working on systems that manipulate large numbers of objects that must be easily stored and retrieved based on their actual contents, you are sure to find in hash tables one of your most consistently useful tools.

10.8 DISPENSERS

Arrays and hash tables are *indexed* structures:

- When inserting an item, you give some identifying information, such as the index in an array and the key in a hash table.
- To access an item, you must provide the associated key or index.

The next (and last) structures we'll study follow a different policy. There's no key or other identifying information for items; you'll insert an item just by itself, typically through a procedure

```
put (x: G)
    -- Add x to current structure.
```

then to retrieve an item you don't have any influence on which one you'll get; the basic query is

```
item: G
    -- Item obtained from current structure.
require
    not is_empty
```

with no argument (compare with *item (i: INTEGER): G* for arrays and *item (k: KEY): G* for hash tables). We call such structures **dispensers**, by analogy with a simple vending machine as illustrated: the provider loads the machine with cans of soft drinks; after putting a coin, the customer will get a can — *any* can — from those in the machine. The machine, not the customer, chooses which can to deliver if more than one is available.



A dispenser

Dispensers differ in the policy the machine uses to select the item to deliver:

- Last-In First-Out: choose the element inserted most recently. A dispenser with a LIFO policy is called a **stack**.
- First-In First-Out: choose the oldest element not yet removed. A dispenser with a FIFO policy is called a **queue**.
- With a **priority queue**, items are assumed to have an associated “priority” (an integer or real number); the query item will return the element with highest priority. Although this case seems closer to indexed structures, it’s still an example of dispenser, as the priority is an intrinsic property of each item, rather than information added for storing it into the data structure.

For all dispensers, the four basic features are *put* and *item* with the signatures and precondition shown above, the boolean query

```
is_empty: BOOLEAN
-- Are there no items?
```

and a command to remove an element:

```
remove
-- Remove item from current structure.
require
not is_empty
```

Just as *item* doesn’t let you choose which element to access, *remove* doesn’t let you choose which element to remove; but as the comment indicates, the element removed is the one that *item*, called just before, would have yielded.

A good implementation of dispensers should make all these operations execute in constant ($O(1)$) time; we’ll see examples below.

In some libraries you will find an operation that combines the effect of *remove* and *item*: it’s a function, say *get*, that removes an item, and return as its result the value of that item. We could implement such a function in terms of *remove* and *item*:

```
get: G is
-- Side-effect-producing function, violates style rules!
do
  Result := item
  remove
end
```

We won't use any such function, however, since it violates the Command-Query Separation principle, by both changing the structure and returning a result. For reasons explained in an earlier chapter, it is preferable to let clients access and remove items through two separate features.

The next two sections covers stacks and queues. We won't examine priority queues in detail, but you may study the EiffelBase class *PRIORITY_QUEUE*.

10.9 STACKS

A stack is a dispenser applying a LIFO policy: the item that you can access at any given time is the one added most recently. The place of access is called the “top” of the stack, and indeed the natural image is that of a stack in the ordinary sense, for example the set of dictionaries on my desk, assuming I can only pick the top element:



A stack

The “Towers of Hanoi” studied in a later chapter to illustrate recursion also function as stacks.

→ See the figure on page [365](#).

Another visual illustration, more related to the notion of dispenser, is a piggybank-like device where you insert and retrieve coins at the same end:

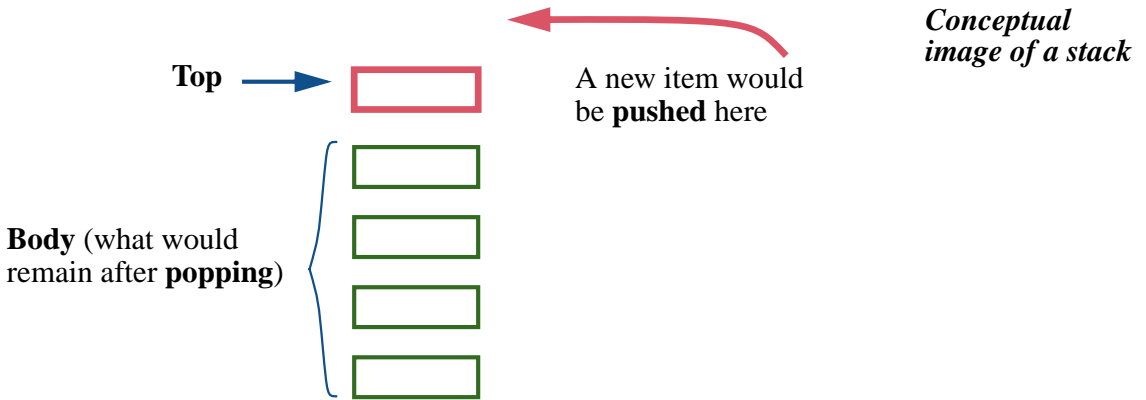
A stack

Stack basics

The stack operations are often known as:

- **Push** an item to the top of the stack (command *put*).
- **Pop** the top item (command *remove*).
- Access the **top** element (query *item*).

which you may visualize as this:



Using stacks

Stacks have many applications in computer science. Here are two examples. Assume you want to evaluate a mathematical expression in **Polish notation**, a parenthesis form often used by pocket calculators, or as internal form by a compiler or interpreter, because it is unambiguous without needing to use parentheses: each operator applies to the previous two operands, and the result of evaluating defines an operand for the next operator. For example the expression $2 + (a + b) * (c - d)$ is represented in Polish notation as

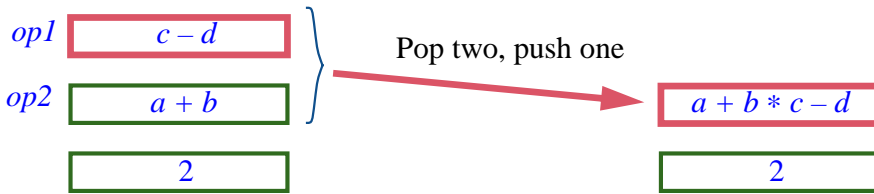
$$2 a b + c d - * +$$

with the following meaning, corresponding to the intended value: the first $+$ applies to the previous two operands, leading to the new operand $a + b$; the $-$ operators applies to the previous two operands, leading to the new operand $a - b$; then the $*$ applies to these two resulting operands, leading to the new operand $(a + b) * (c - d)$; the final $+$ yields the sum of 2 (the first of all operands) and this result. The following algorithm, using a stack of operands s , evaluates a general Polish expression:


```

from
until
    "All terms of Polish expression have been read"
loop
    "Read next term x in Polish expression"
    if "x is an operand" then
        s.put (x)
    else -- x is a binary operator
        -- Obtain and pop the two top operands:
        op1 := s.item; s.remove
        op2 := s.item; s.remove
        -- Apply operator to operands and push result:
        s.put (application (x, op1, op2))
    end
end
    
```

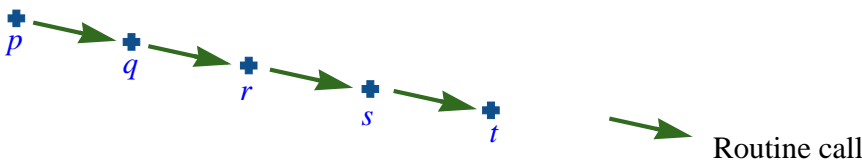
This uses two local variables *op1* and *op2* representing operands, and assumes a function *application* that yields the result of applying a binary operator to operands, for example *application* ('+', 2, 3) is 5. The following figure shows the algorithm's key operation, as expressed by the **else** clause, at the time of processing the * operator in the example expression.



Polish expression evaluation

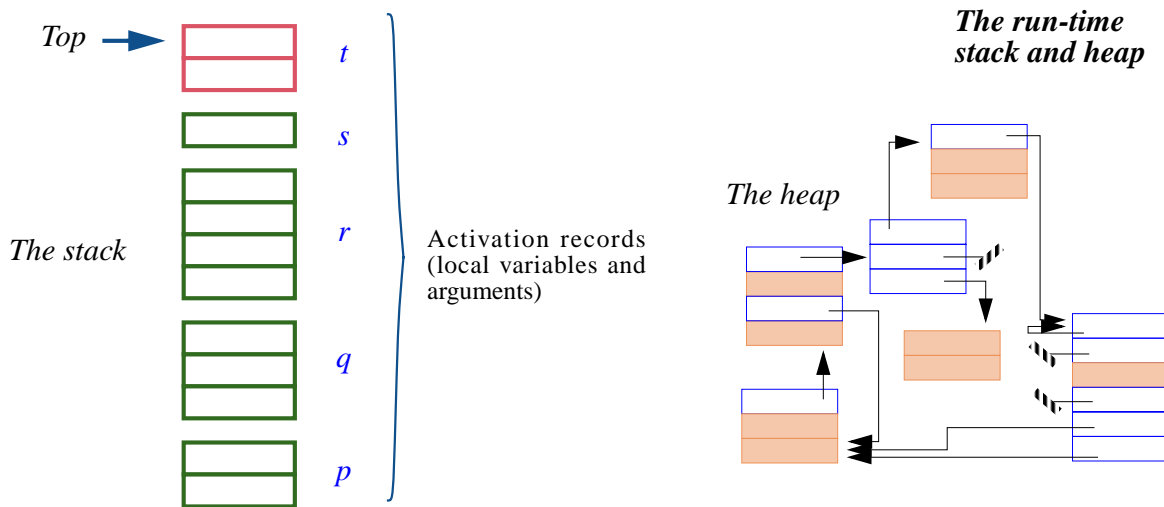
A proper implementation of the algorithm must handle erroneous input (by checking for *s.is_empty* before using *item* and *remove*, and checking in the **else** clause that *x* is an operator), and consider unary as well as binary operators.

Our second example underlies *every* modern programming language implementation and is present in every operating system (that's a strong statement, but I can't think of a counter-example). Consider a programming language such as Eiffel, where a routine can call a routine, which can call a routine, which; this is known as the **call chain**:



Call chain

At any run-time moment several routines — p to t in the figure — have been started and not yet finished; the last one that started, here t , is the “**current routine**”. Consider any of its instructions, say an assignment $x := y + z$. Unless the entities x, y, z are attributes of the enclosing class, they must belong to the current routine, either as **arguments** (except x , since we can’t assign to arguments) or **local variables**. Let’s use the term “locals of the current routine” for both categories. To perform instructions such as this assignment, the code generated by the compiler must have access to all the locals. The solution is, on every routine call, to create an **activation record** containing the routine’s locals:



The structure on the right is the *heap*, which contains the objects allocated through **create** instructions or equivalent. Of interest for the present discussion is the *run-time stack* (or just “The Stack”), containing the activation records for all currently active routines. Because no routine execution terminates until the execution of all the routines it has started terminates, the routine activation scheme is LIFO, and a stack is the appropriate structure.

In many programming languages routine texts can be *nested* (enclosed in others); then an instruction may refer not only to locals of the current routine but also to locals of any enclosing routine. This means that the execution may need access not only to the top activation record, but also to a few others below it. In this scheme the activation record structure — still called “the stack” — uses a slightly extended notion of stack. Eiffel doesn’t need routine nesting.

On routine entry, the mechanism creates a new activation record for the routine (with the local variable entries set to the default initial values, and the argument entries set to the values of the actual arguments for the call) and pushes it onto the stack. On routine return, it pops the stack.

A benefit of using a run-time stack is that the activation records are not required to represent different routines — only different routine *executions*. As a result, this technique supports **recursive** routines: routines that call themselves, directly or indirectly. Allocating a new activation record for every new call allows each recursive call to use its own set of locals, distinct from any locals used by previous, still active incarnations of the same routine, which have their own activation records further down in the stack. Recursion is the topic of an entire chapter, and its implementation — based largely on stacks — of one of the chapter’s sections.

→ “[THE IMPLEMENTATION OF RECURSIVE ROUTINES](#)”, [16.7, page 396](#).

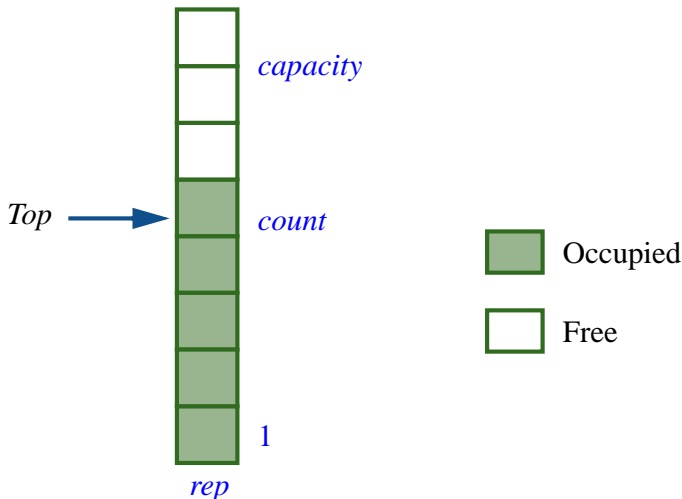
Implementing stacks

As with several other structures of this chapter, there are two general categories of stack implementations: arrayed and linked.

By far the most common implementation uses an array *rep* of type `ARRAY [G]` and an integer *count*, with the invariant

$count \geq 0$
 $count \leq rep.capacity$

where *capacity* is the number of array items ($upper - lower + 1$). With the array indexed from one ($lower = 1$), the stack items if any are stored in positions 1 to *count* of the array: ← See page [258](#).



**Arrayed
implementation
of a stack**

In class `ARRAY` the number of items is known as both *count* and *capacity*, with an invariant stating they are equal. This should not be confused with the *count* of stacks, which gives the number of stack items — in the arrayed implementation, the number of array positions occupied by stack elements.

In this implementation, the query *item* giving the top item simply returns *rep.item(count)* the array item at position *count*; the command *remove* can be implemented as simply *count := count - 1*, and *put(x)* as

```
count := count + 1
rep.force(x, count) [11]
```

using the command *force* of arrays that will perform a resizing if *count* outgrows the current array *capacity*.

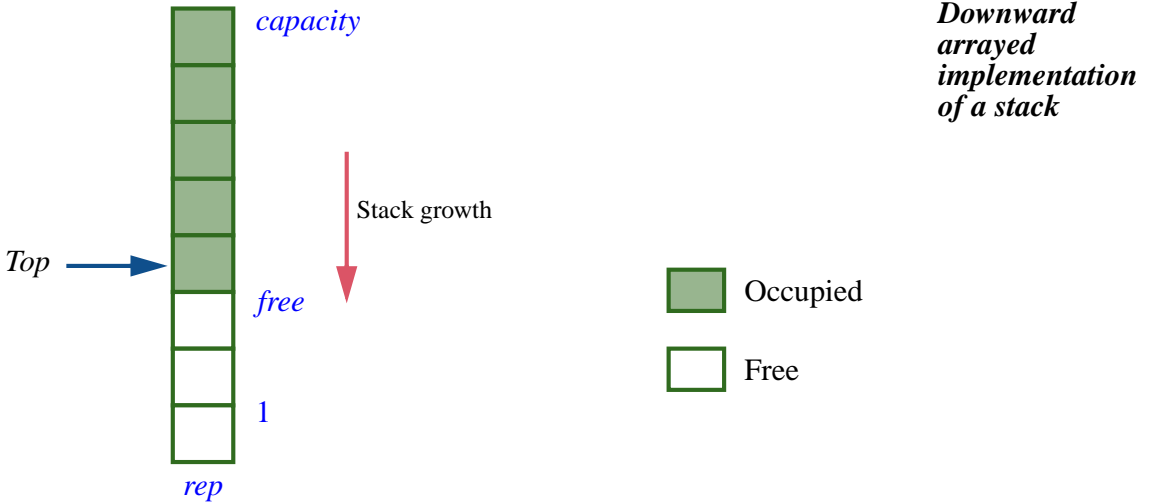
This implementation is what you'll find in the EiffelBase class *ARRAYED_STACK*. (This class does not actually need *rep* since it *inherits* from class *ARRAY*, but this is conceptually equivalent and we haven't studied inheritance yet.) The use of *force* for the algorithm of *put* means you don't have to worry about dimensioning the array properly; the array just starts out with a default size and gets resized as needed when you push items.

Of course, the available memory is limited in the end, so you still have to ensure the total size of your data structures remains within control.

Array resizing is not commonly available outside of Eiffel, so most other arrayed implementations of stacks use a fixed-size array, which may be inevitable in some cases anyway (even in Eiffel) if you need to control memory tightly. The corresponding EiffelBase class is *BOUNDED_STACK*. For a bounded stack there is, along with *count*, a query *capacity* (implemented in the arrayed representation as *rep.capacity*) and a boolean query *is_full*, whose value is *count = capacity*. Then in the same way that *remove* has the precondition *is_empty*, the command *put* now has the precondition **not is_full**, ← Page 292. and its array implementation (see <ww-redonly>[11<ww-redonly>] above) uses *put* rather than *force*; this is correct since ensuring **not is_full** will guarantee that the call to *rep.put* satisfies the precondition *valid_index(count)* ← Page 260. of *put*, which here means that *count* must be between 1 and *capacity* inclusive.

All the operations cited are **O(1)** in time.

A variant of the arrayed representation, bounded or not, has the stack grow downward:

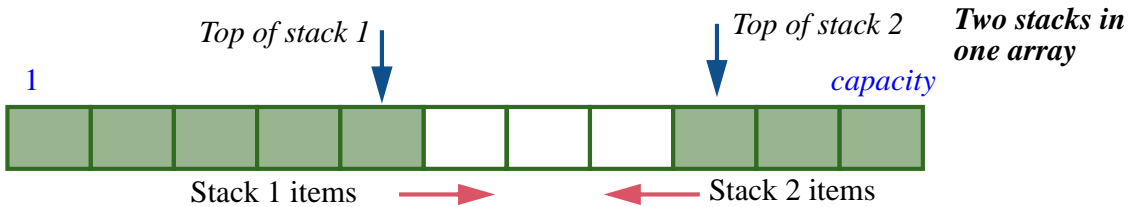


In this representation, *count* is no longer an attribute; it's replaced by a secret attribute *free* giving the index of the first free position. Query *count* must still be available, of course; it is a function that returns $capacity - free$. The invariant will state that $free \geq -1$ and $free \leq capacity$ (compare with the requirement on *count* in the previous representation). The case $free = -1$ corresponds to *is_full*, and $free = capacity$ to *is_empty*; the items, if any, are in positions *capacity* down to $free + 1$. The implementation of *remove* is $free := free + 1$, and the implementation of *push* is

```

rep.force (x, free)           -- A bounded version will use put
free := free - 1
    
```

If you have limited space available and *two* stacks, you can store both of them in a single array, using the upward scheme for one and the downward scheme for the other (appearing as rightward and leftward on the following figure):

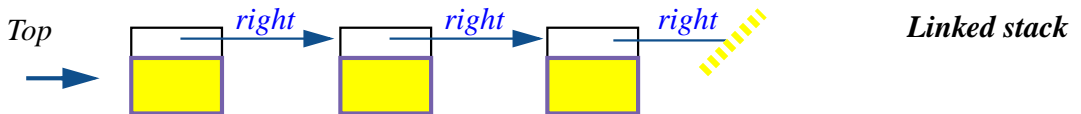


The advantage of this technique over two separate arrays is that it achieves a better use of space if the two stacks do not reach their maximum *count* together. Denoting by $\max(x)$ the maximum value of a mathematical variable x , we note that

$$\max(\text{count1} + \text{count2}) \leq \max(\text{count1}) + \max(\text{count2})$$

so that a one-array representation of size $2 * n$ might still have space available if one of the stacks has more than n items, whereas with two arrays of size n we run out of space as soon as either stack reaches n . An exercise asks you to write a class *TWO_STACK* implementing this idea. → [10-E.2, page 308](#).

Along with arrayed implementations, you can use a linked representation for stacks. Indeed a linked list as studied earlier in this chapter provides a ready-made implementation of a stack, as shown in the next figure: the first cell is the top, the rest of the list is the stack body.



The operation *put(x)* is implemented simply as *rep.put_front(x)*, where *rep* is the linked list; *item* is just *rep.first* (where *first* for linked lists yields the first element, *i*-th (1)); and so on. Class *LINKED_STACK*, in EiffelBase, provides such an implementation. The basic operations are still $O(1)$, although significantly slower than in the arrayed versions; for example, *put_front* of *LINKED_LIST* and hence *put* of *LINKED_STACK* must allocate a new *LINKABLE* cell.

All the basic operations are indeed constant-time in the various implementations of stacks we have seen with, as noted, the occasional exception of a call to *force* in a resizable stack:

Operation	Feature in stack classes	Complexity	Comments
Access top	<i>item</i>	$O(1)$	
Push to top	<i>put</i>	$O(1)$	With automatic resizing, occasionally $O(\text{count})$
Pop	<i>remove</i>	$O(1)$	

10.10 QUEUES

With their First-In First-Out policy, queues are useful in many applications. Two typical examples:

- In *simulation* applications, especially the variant known as *discrete-event* simulation, a program perform steps simulating what's happening in some process — a assembly line producing cars from parts, a network transmitting messages, a store serving customers — to analyze waiting times and remove inefficiencies. Often, the handling of events (parts arriving on the assembly line for processing, messages arriving on the network, customers arriving at the store) is first-in first-out; a queue will represent the pending events.
- A similar situation arises in a Graphical User Interface (GUI) system, where the events triggered by users — mouse clicks, cursor movements, key presses — should be processed in the order of arrival.
- In operating systems and other cases of concurrent programming, a frequently useful scheme is *producer-consumer* communication where one process, the producer, generates some information, which another, the consumer, reads and processes in the order of production. The structure used for the exchange of information is a queue, in a variant known as the **buffer**, adapted for concurrent processing.

Producer deposits



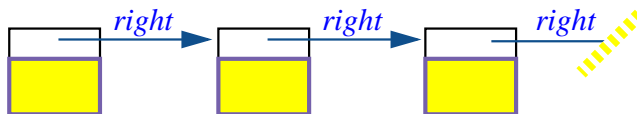
Consumer-producer communication through a buffer

Consumer accesses and remove items

The last figure can serve as a conceptual representation of any queue, not just a buffer: insert items at one conceptual end, remove them at the other end.

As with stacks, we may use linked and arrayed representations. A linked implementation is straightforward:

Insert here



Linked queue

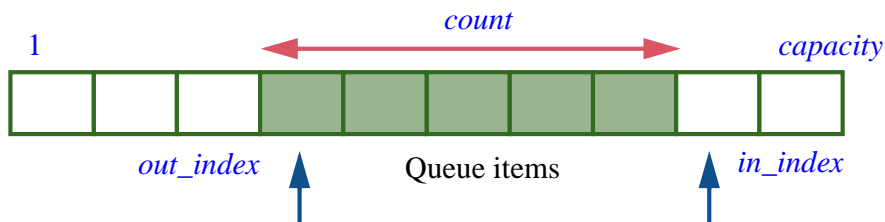
Access and remove here

For example the operation *put* (*v*) will just be *rep.put_front* (*v*) (if as usual *rep* is the implementing structure, here a linked list), *item* must return the last item of the list, and *remove* must remove it. The EiffelBase implementation, class *LINKED_LIST*, maintains the invariant

is_always_after: **not empty implies rep.after**

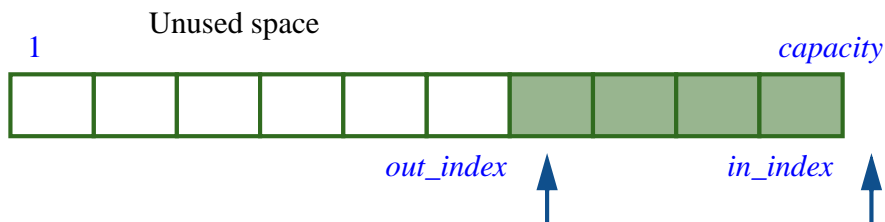
meaning that the list cursor is always past the last item.

The arrayed representation is a little more tricky than for stacks because we must remove elements at one end and insert new ones at the other. Instead of just one integer marker *count* we should maintain two, which the class *ARRAYED_QUEUE* calls *in_index* and *out_index*, both secret attributes. (The public query *count* is still there, giving the number of items.) It's not good enough, however, to use the interval *in_index..out_index* to store items, as in this simple picture



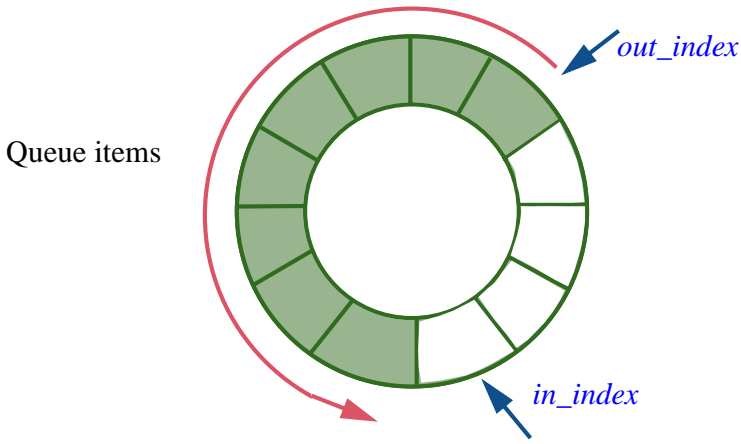
A possible state for an arrayed queue

(with the obvious implementation of *remove* as *out_index := out_index + 1* and *put* (*x*) as *in_index := in_index + 1*; *rep.put* (*x*, *in_index*, where *rep* is an array), since we would quickly run out of space after a few *put* even if, as a result of one or more *remove* there's unused space at the beginning of the array:



Arrayed queue reaching right end of array

The solution: when the *in_index* marker reaches past *capacity*, the next *put* should make it cycle back to the beginning of the array, and similarly with *remove* for *out_index*. Conceptually it's as if we wrung the array to bring its two ends together, turning it into a ring:



Portrait of the array as a doughnut

Here for example is *put* from *ARRAYED_QUEUE*:

```

put (v: G) is
  -- Add v as newest item.
  do
    if count + 1 = rep.count then grow end
    rep.put (v, in_index)
    in_index := (in_index + 1) \ \ capacity
    if in_index = 0 then in_index := capacity end
  end
    
```

The first instruction reallocates the array if we truly run out of space. (“*Don’t box us in*”). When we increment *in_index* in the highlighted instruction, we do it modulo *capacity*: $i \ \backslash \ \ j$ is the integer remainder of i by j , as $i \ // \ j$ is their integer quotient. The implementation is tuned (see the final *if...*) to an array *rep* indexed from 1 to *capacity*; it’s also possible — and a recommended *exercise* — to see what it gives for an array indexed from zero, and to write the corresponding *remove* implementation.

← “Automatic resizing”, page 251.

→ 10-E.3, page 308.

Queues, with proper representation, yield the same performance as stacks

Operation	Feature in queue classes	Complexity	Comments
Access oldest item	<i>item</i>	$O(1)$	
Add item	<i>put</i>	$O(1)$	With automatic resizing, occasionally $O(count)$
Remove oldest item	<i>remove</i>	$O(1)$	

10.11 OTHER STRUCTURES

The data structures that we have seen are among the most important in programming, but by no means the only ones. We’ve already had a glimpse of *trees* (in the form of abstract syntax trees) and will see more of them in connection with recursion. Trees have many variants, such as binary trees and B-trees. A generalization of trees, useful in many applications (for example networking) is the notion of *graph*, directed or not, and *multigraph*.

The bibliographical section cites books that review the fundamental data structures, usually in connection with fundamental algorithms. In addition a number of textbooks address the “Data Structures and Algorithms” courses offered by most computer science curricula.

10.12 ITERATING ON DATA STRUCTURES

Container data structures such as the ones we’ve studied in this chapter are repositories of objects. A common need, on such structures, it to apply a certain operation repeatedly to all these objects. This is known as *iterating* on a data structure. Here is a general definition:

Definition: Iterator

An iterator is a mechanism that can yield, from one or more operations applicable to individual elements of a container data structure, an operation applicable to the structure as a whole.

A simple example of iterator is the routine that computed the total travel time on a metro line. In this case the “operation on individual elements” updates the travel time, as computed so far, by adding the travel time to the next station. More generally, if *your_list* is a *LINKED_LIST [T]* or more generally a *LIST [T]* (in any implementation of lists) and there is a procedure

← *Function total_time*,
page 215.

some_operation (x: T) ...

the following scheme will *iterate* the operation over the list

```

from
    your_list.start
variant
    your_list.count - your_list.index + 1
invariant
    -- All elements before cursor have been subjected to some_operation
until
    your_list.after
loop
    some_operation (your_list.item)
    your_list.forth
end

```

Alternatively, you may want to apply the operation only to items that satisfy a certain condition. given by a function *your_test (x: T): BOOLEAN*; the loop body then changes to

```

if some_test (your_list.item) then
    some_operation (your_list.item)
end

```

Other variants of iteration include:

- Apply an to all items until the first one that satisfies, or doesn't satisfy, a certain condition.
- Find out if at least one item, or all items, satisfy a condition.

In classes such as *LIST* and *LINKED_LIST* you will find iterator features — coming from their ancestor *LINEAR* — that provide these mechanisms: *do_all*, *do_if*, *do_while*, *do_until*, *for_all*, *there_exists*. They use loops such as the above (as you'll see from a glance at their implementations), so that you don't have to write these loops in your own client software. Indeed you can obtain the effect of the first example, applying an operation to all items, simply by writing, in our two examples

```

your_list.do_all (agent some_operation)
your_list.do_if (agent some_operation, agent some_test).

```

We haven't seen the **agent** notation and indeed to understand the interface of features such as *do_all* we'll need a little more baggage. In a nutshell, such iterator features need an argument representing an operation or test. A routine such as *some_operation* can't directly serve as argument because it denotes code, not an object. An *agent* is precisely an object that is associated with a feature; **agent** *some_operation* represents the routine *some_operation* and so can be passed as argument to iterators such as *do_all* and *do_if*. To understand the details you'll need to wait until the chapter on agents; but you can already use simple iterator schemes as in these two examples, by providing features with the right signatures, for a list or similar structure whose items are of a type *T*: procedures such as *some_operation*, taking a single argument of type *T*; functions such as *some_test*, taking an argument of type *T* and returning a boolean value. → Chapter 20.

10.13 FURTHER READING

Donald W. Knuth: *Fundamental Algorithms*, volumes 1 (*Fundamental Algorithms*) and 3 (*Sorting and Searching*) of *The Art of Computer Programming*, 3rd edition; Addison-Wesley, 1997.

Widely considered the ultimate reference on algorithms and data structures. Part of a planned 7-volume set of which only 3 have appeared.

Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

A compact survey of the most important algorithms and data structures. Still an excellent survey of the field, suitable after a first introduction as given in this chapter.

Bertrand Meyer: *Reusable Software*, Prentice Hall, 1994.

A presentation of design principles for building quality reusable libraries, illustrated through the example of EiffelBase.

10.14 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Static typing makes program clearer and makes it possible to catch errors at compile time.
- A generic class has one or more parameters representing types. This provides an extra degree of flexibility and is particularly useful for classes describing container structures.
- Data structures should support resizing.
- For the consistency of a library, it is desirable to stick to a standard feature naming policy.
- Abstract complexity estimates the performance of algorithms, independent of hardware choices, by focusing on algorithm behavior for large data sizes and ignoring constant multiplicative and additive factors.
- “Big-O” notation, as in $O(n^2)$, expresses abstract complexity.
- Arrays provide fast, constant-time access and replacement of items known through their indexes in a given range. Although they can be reallocated, they are not suited for the representation of structures with frequent item insertions and deletions.
- Hash tables generalize arrays to indexes that can be almost arbitrary “keys”, for example strings (not just integers), while keeping access and replacement time essentially constant.
- Lists, especially in linked representation, serve to describe sequential structures, and support insertions and deletions.
- Dispensers let you access, insert and remove elements at only one place. The policy can be Last-In First-Out, yielding stacks, or First-In First-Out, yielding queues.
- Stacks are particularly useful to represent nested structures and have applications throughout operating systems and compilers. An array implementation is the most common; a single array can huse two stacks.
- Queues are particularly useful in modeling, and in concurrent programming as “buffers”. With an array implementation, array indexes should cycle past the upper bound after repeated insertions and deletions.

New vocabulary

Abstract complexity	Activation record	Actual generic parameter
Array	Call chain	Complexity
Correctness	Cursor	Dispenser
Dynamic typing	FIFO	Formal generic parameter
Generic class	Generic derivation	Genericity
Hash table	Heap	Linked list
LIFO	List	Parameter
Priority queue	Queue	Run-time stack
Stack	Static typing	Validity

10-E EXERCISES

10-E.1 Vocabulary

Give a precise definition of all the terms in the above “New vocabulary” list.

10-E.2 Two in one

Write a class *DOUBLE_STACK* [G] implementing two stacks in a single array; you may call the features *put_1*, *put_2*, *remove_1*, *remove_2* and so on. ← Page 260.
The stacks are of bounded size; make sure to include the right preconditions and invariant clauses.

10-E.3 Indexing from zero

The implementation of arrayed queues, drawn from the EiffelBase class *ARRAYED_QUEUE*, uses an array *rep* indexed from one. ← Page 303.

- Using for inspiration the implementation of *put* given in the text, write the routine *remove*, and a creation procedure *make* setting up the queue as empty of any items.
- Rewrite all three routines so that the implementation array is indexed from zero.

11

Input, output and exceptions

11.1 READING FROM A FILE

11.2 WRITING OUT

11.3 ABNORMAL CASES: WHEN THE CONTRACT IS BROKEN

11.4 RECOVERING FROM EXCEPTIONS

PART II:

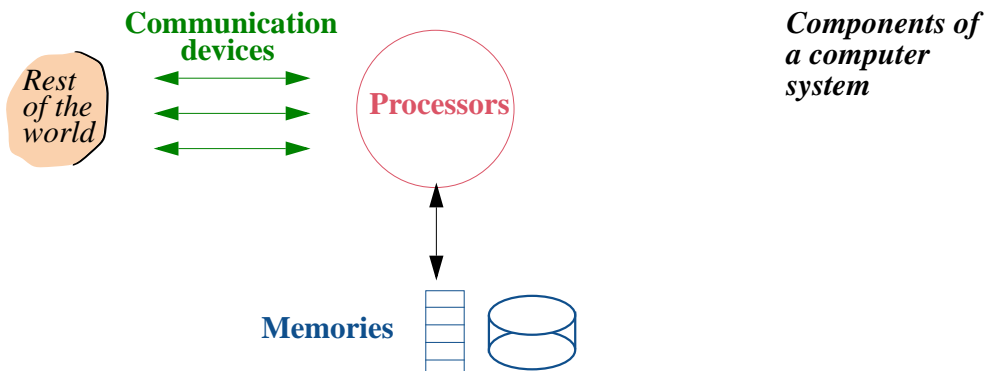
How things work

We start with the essentials of programming: objects, classes, interfaces and contracts, and supporting concepts including logic and some elements about hardware.

12

Just enough hardware

There would be no software without computers. To understand what it takes to develop good programs, we must have a basic understanding of the underlying machinery — the hardware — on which they will run. In this chapter we'll take a look at some of the essentials of what you must know about that hardware, detailing some elements of our earlier overall picture:



In particular, we'll try to get a feel for the *order of magnitude* of hardware phenomena: how much information you can represent through computer data, how fast you can access such data and execute operations on it.

We limit ourselves to properties of direct relevance to programmers and to the topics in the rest of this book. Along with learning to program, you should be taking a course on a topic such as Introduction to Computer Architecture, which will go far deeper into the details.

12.1 CODING DATA

The data that we store in our computers' memories represents information of very diverse nature, from employee records, images and sounds, texts in human languages with formatting information (fonts, layout), to numerical values used in scientific computation — not to forget programs. We need a general way to represent this information and interpret the corresponding data.

The binary system

Part of what made the computer revolution possible was the discovery of a simple and general way to represent information as data: the binary system.

The basis of the binary system is a set of two values (hence “binary”). These values have no intrinsic meaning, so we might call them Black and White, , Tom and Jerry or maybe Isis and Osiris. What matters is that they be unambiguously different. In fact we call them 0 and 1 (zero and one).

The term **bit** denotes a mathematical variable whose possible values are just these two. The word was made up by engineers in the late 1940s as a contraction of *binary digit*, to indicate that a bit is like a digit of ordinary arithmetic (0, 1, ... 9) but with 0 and 1 the only possible values.

A bit (low-tech version)

“Bit” also denotes, by extension, a physical device that has two possible states, and hence can be used to represent a mathematical bit once we agree on which will be one and which zero. A cardboard sign on your door with a little flag that you can move to read either “The doctor is IN” or “The doctor is OUT” is a bit. More relevant to the computer industry are electronic bits (obtained for example by transistors), where the two states correspond to two different voltages, and magnetic bits, for example small areas of magnetic tape or disk, where the states are magnetized and unmagnetized.

The reason why the binary system works so well is that today’s electronics industry lets us:

- Build such physical bit representations and pack many of them in a small area. To be more precise: pack *very* many of them in a *very* small area. We’ll see some figures below.
- “Write” these bits (change their values) and “read” them (obtain their values) quickly. Very quickly.
- Build many such collections of bits, cheaply. Very cheaply.

These properties have ensured the success of the binary system. Some early computers used a *decimal* system; this seemed more natural since computers were largely then seen as counting machines, and when people count they will probably — computers or not — continue to use a decimal system for a long time, if only because we have ten fingers, not two or eight or sixteen. (The word *digit* itself comes from the Latin for “finger”.) But for automatic computers built with the devices of electronics, the binary system long ago displaced all its competitors.

To what extent is this relevant to programmers? More than you might think at first. True, we work on programs in their source form expressed in a pleasant programming language, where the connection to *information* is clear, so that we write numbers, for example, in the usual decimal notation: 10, or -1, or 3.1415926524. But as soon as we consider how data is represented in memory, and in particular *where* it is stored, we’ll have to remember that the basic numbering system is the one that’s natural for computers and strange to people, rather than the other way around.

That’s why we’ll take a look now at some of the properties of the binary system and its associates, although this is not a substitute for the more detailed knowledge you’ll gain from courses on logic and digital design.

Binary basics

Unless all the information you ever deal with is the result of a single toss of a coin, two possibilities isn't much. The basic combinations, out of which you can encode finite data of any size, are:

- The **byte**, or sequence of *eight* bits.
- The **word**, usually denoting a sequence of four bytes, or *thirty-two* bits.

Early on, the definition of “word” was not so universal; some computers used different word lengths. You can still encounter such cases, but they are rare. Bytes have always been 8 bits and are also called *octets*.

How many possible values can such a sequence of bits represent? One bit has two possible values, 0 and 1. Put two bits together; for their combined values there are four possibilities:

0	0
0	1
1	0
1	1

More generally for a sequence of n bits for any integer $n > 0$ there are 2^n (two to the power n) possibilities, as is easily proved by induction.

Basic representations and addresses

For the basic units:

- A byte, with eight bits, has 256 (2^8) possible values.
- A collection of 32 bits has 2^{32} possible values; that number, given in the table below, is on the order of four billions.

For example if we want to store *characters* making up a text, we'll use one **byte** for each character. 256 possibilities might seem a luxury, but in fact it's just about what we need once we have included the ten digits, special symbols on your keyboard — `~`, `!`, `@` etc. —, the 26 lower-case and 26 -upper-case letters of the Roman alphabet, and the most common accented letters of Western languages (é, Ä and so on). The standard assignment of each possible 8-bit configuration to represent each one of these characters is known as *extended ASCII*; the original ASCII used only 7 bits (128 possibilities) and had no support for accented letters.

“American Standard Code for Information Interchange”. You may see the ASCII byte code assignments at www.asciitable.com.

For languages with other character sets, such as Cyrillic, or ideograms, such as Chinese, extended ASCII doesn't suffice any more; the standard there is *Unicode*, which uses two or more commonly four bytes for a single character, supporting a large set of possibilities that cover the most important written languages.

Our programs will use the name *CHARACTER* to denote the type of value that can be stored to represent a character; depending on a configurable setting this may be either extended ASCII or Unicode.

For *numeric* information, the common practice is to use a **word** to represent an integer variable. The mathematical set of integers is of course infinite, but in the memory of a computer we'll have room for only a finite set of values; using a word, we can represent about two billion negative values and two billion positive ones, which is usually enough. The type for such data, in our programs, will be written *INTEGER*.

A word can also serve to represent non-integer numerical values, mathematically corresponding to *rationals*, such as $3/2$, and *reals*, such as π . Such values are particularly useful in “scientific computation”, the use of computers for solving problems with a strong numerical component in physics, biology, engineering or even finance. The corresponding type in our programs is called *REAL*; it uses a representation of non-integer values known as “floating-point”. For many applications, however, 2^{32} as provided by a word doesn't give us any precision in modeling the real numbers; the type *DOUBLE* uses two words, with 2^{64} possible combined values, and is usually the one you need for serious numerical computations.

The starting position at which a data element appears in memory is called its **address**. The example of data types *CHARACTER*, *INTEGER*, *REAL* and *DOUBLE* indicates that data elements may be of different sizes (in these cases one byte, four bytes, four again and eight). To provide a uniform way of denoting addresses, the convention is always to count in bytes, and to start at zero (rather than one). So if the memory starts with a thousand values of type *INTEGER*, the first element that follows them will be at address **4000**.

Powers of two

If only because of the property just seen (n bits can have 2^n values), the powers of 2 are important to the binary system. Here are the first ten and other important ones:

n	2^n	Approximation by power of 10	Common name (abbreviation)
0	0		
1	1		
2	4		
3	8		
4	16		
5	32		
6	64		
7	128		
8	256		
9	512		
10	1024	10^3 (thousand)	Kilo (K)
16	65536		
20	1,048,576	10^6 (million)	Mega (M)
30	1,073,741,824	10^9 (billion)	Giga (G)
32	4,294,967,296	4×10^9 (4 billions)	
40		10^{12} (trillion)	Tera (T)
50		10^{15}	Peta (T)

You need to remember the first ten values, the order of magnitude of the others listed, and the abbreviations (kilo etc.).

From cherries to bytes

In the ordinary, decimal way of counting things, abbreviations like “kilo” represent powers of ten, more precisely the powers of 10^3 which serve as natural milestones: a kilogram of cherries at the market is one thousand grams (10^3), one million dollars (10^6) won’t buy you anything decent in Southern California, one billion Swiss francs (10^9) might prolong the life of a failing airline by a few weeks.

This also applies to computer-related measurements other than memory:

- A transmission line functioning at 1 Mbps (Megabit per second) can transmit one million bits each second.
- A CPU with a speed of 1GHz (one Gigahertz) can execute one billion basic processor instructions per second. “Hertz”, number of events per second, is a frequency measure borrowed from physics.

While memory sizes and addresses are expressed in *bytes* (abbreviation **B**), transmission speeds are usually given in *bits* per second or **bps**, where the abbreviation for “bit” is **b**. So a “56K modem” — if functioning at its highest rate, which it usually doesn’t — would transmit 56,000 bits each second.

To express memory size, computer engineers prefer to use the powers of two.

To connect the two systems, decimal and binary, we note in the preceding table that the tenth power of two, 2^{10} , is **1024**, slightly over 10^3 , a thousand. As a consequence, 2^{20} is of the same order of magnitude as a million, 10^6 , and 2^{30} as a billion, 10^9 ; see the exact values in the table. This explains the last column: for memory measurement we reuse the common abbreviations of decimal powers, but to represent the closest *binary* powers.

To avoid any confusion, remember that the binary interpretations are only used for *memory* measurements. For anything else the usual decimal meanings continue to apply. If the ad for that 1-GHz laptop, which executes a billion operations per second, also says it has 1 GB of memory, you will actually be getting more than a billion bytes; about 73 million more.

In most practical cases the difference doesn’t matter: between friends, what’s a few millions?

“Real” numbers

[TO be completed] More precisely, *REAL* describes a finite set of values — a large set, with as many as 2^{32} or 2^{64} values, but still finite —, so:

- If x is a mathematical real number, what we can store on the computer is the closest to x of these values; this introduces a small error.

- If x and y are two mathematical numbers and x' and y' the values associated with them on the computer, the programming language notation $x + y$ does not denote the mathematical sum of x and y ; it generally does not *even* represent the exact value of the mathematical sum $x' + y'$, since that value may not be directly representable on the machine; it's only a close approximation of it. The same holds for other arithmetic operations. What this means in practice is that every application of such an operation may introduce a small numerical **error**.

This is a typical example of the distinction between *information* and *data*.

← “Definitions: Data, information”, page 10.

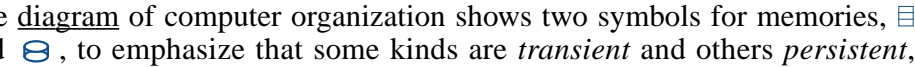

In computations using real numbers — “*scientific computation*” for science and engineering but also, for example, financial modeling — approximations are usually acceptable; the problem is that in an actual computation performing millions or billions of elementary operations the errors, individually small, may accumulate to the point where they seriously affect the validity of the results. Numerical programming requires careful techniques to fight this phenomenon; we’ll encounter a typical one in the example of an integration routine, leading to an important methodology principle.

→ “AGENTS FOR NUMERICAL PROGRAMMING”, 19.4, page 473.

12.2 MORE ON MEMORY

Memory is where we put and access the data. At the most elementary level it holds basic data elements such as characters and integers, but to our programs it will be the place where we create and find **objects**. Let’s see what memories can do for us.

Persistence

The diagram of computer organization shows two symbols for memories,  and , to emphasize that some kinds are *transient* and others *persistent*, supporting data with different requirements:

← Page 313.

- Transient data is created and manipulated by a program’s execution, but is not guaranteed to survive that execution. With some memory technologies, powering off the memory unit will lose the data.
- Persistent data remains forever unless expressly deleted; switching off power has no effect on this property.

Why have transient data at all? It might be simpler to make all data persistent by default, and delete what we don’t need any more. The answer is technological and economic. Memories that processors can access at a reasonable speed for their data-processing operations are transient, and expensive; persistent memories are cheaper, so that we can (thankfully) use them to store large amounts of data — representing text, images, music, flight tables, personal information ... — which we’ll have to access more slowly.

Words like “reasonable” or “slower” speed, “large” or smaller amounts of data, “cheap” or not, should be put in context. As rough estimates in today’s technology:

- A good computer for software development, possibly a laptop, might have a transient memory capable of holding one to two GB (gigabytes) at a cost of 200 dollars or euros for the memory. The time to access a character might around 10 nanoseconds, meaning 100 million accesses per second. (A nanosecond or ns is 10^{-9} seconds.)
- The computer might have a **disk** (persistent memory) that is sixty times as big (60 GB), costs twice as little (\$100), and has an average access time on the order of a millisecond — a thousand accesses per second.

The ratio of access times is remarkable. If we scaled up to human-level magnitudes, the program would be like a worker who has some papers accessible on a shelf within reach of hand, say within one meter (three feet), and a much larger archive, almost unbounded for practical purposes, in a storage room *one hundred kilometers* away (100.000 times as far), reachable only by walking. Clearly, the choice of what you keep in your office and what you store away is going to have some effect on your productivity.

This property is directly relevant to the programmer. Programs that manipulate large amounts of data cannot ignore the issue of their distribution between transient and persistent memory; they must keep transfer times under control, for fear of damaging execution speed.

Transient memory

Processor operations, as noted, will access and modify data in transient memory. This key component of computations has several names:

- *Main* memory.
- *Primary* memory.
- RAM, for *Random Access Memory*.

The term has a historical origin: initially, non-primary memory was implemented with technologies such as magnetic tape or, even earlier, *punched tape* (paper tape with holes to represent bits, where the presence of a hole can mean 1 and its absence 0; in such cases data access is *sequential* in the sense that elements are accessed one after the other, in the order of their appearance on the tape. Starting from the beginning, the time to access an element is proportional to its address (since you must first go over all the ones before it). In contrast, main memory is “random”, meaning that it takes the same time to access any element, regardless of its address. Many non-primary memories, such as disks presented below, are now random as well, but RAM has stuck.

- *Core* memory, or just “core”. This term points back to an older technology, little magnetic elements or “ferrite cores” of many years ago, but you’ll still hear that a certain set of data is “in core”. Just understand “core” as meaning central, as in “core competence”.

The photograph below shows a main memory “chip” containing xxx bytes. Its access time is yyy.

A memory chip

Persistent memory

Persistent memory really consists of two kinds:

- Some elements are intended to remain attached to a computer during its operation. They are called **secondary memory** to emphasize that they really serve as extension of the primary memory, with cheaper costs per gigabyte and hence more space available, slower access, and persistence.
- Others are meant to be connected to a computer only episodically so that data can be copied onto them, then removed from the computer, and later on connected again to the same computer or another, enabling reading the data back. So they serve as devices for data “backup” (long-term preservation and storage) and interchange. They are called **removable memory**, or removable storage devices. “*Storage*” is just a synonym for memory.)

The most common form of secondary memory is the **disk**. A more correct term is “disk device” since “the disk” on your computer is actually a pile of magnetized disks, all rotating a speed of 4,000 to 12,000 runs per minute, with reading heads that can move back and forth over disk surfaces to access the data, a bit being represented by the magnetized or demagnetized state of a tiny area. If power is switched off the heads obviously won’t work, but magnetization isn’t lost, making disks suitable for persistent data.

A disk

The disk device pictured above has xx disks and can store yyy gigabytes. It's access time is xxx ms (milliseconds). Note that this is only an average; one of the characteristics of disk access is that there is a *latency* time necessary to get the head to the right position; after that, you can access even a large amount of data much faster if it's all contiguous. When writing programs that mae heavy use of disk data you may have to take this property into consideration for optimized performance.

Although technologies other than disk, such as semiconductor memories, are available for secondary memory, disk devices remain the dominant kind.

There are many kinds of removable memory. Disk devices themselves are often removable. Others include “USB memory sticks”, which connect to the “Universal Serial Bus” connector on a personal computer; the USB stick pictured below can hold 128 MB. Next to it is a removable disk device that connects to another port available on laptops, the “PCMCIA” port, . “ZIP drives”, and the old paper tapes mentioned earlier are other examples of removable storage devices.

For “Personal Computer Memory Card International Association” although you may prefer the unofficial version: “People Can’t Memorize Computer Industry Acronyms”

12.3 COMPUTER INSTRUCTIONS

12.4 GETTING A CONCRETE FEEL FOR COMPUTERS’ POWER

12.5 MOORE’S “LAW” AND THE EVOLUTION OF COMPUTERS

12.6 KEY CONCEPTS LEARNED IN THIS CHAPTER

•

*A memory stick
and a PCMCIA
disk*

New vocabulary

Address	Bit	Byte
Core	Disk	Kilo
Giga	Mega	Persistent
Primary memory	RAM	Read
Removable memory	Secondary memory	Storage
Transient	Word	Write

12-E EXERCISES

12-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

12-E.2 Measurements

How many bytes *exactly* is:

- 1 • One kilobyte
- 2 • One megabyte
- 3 • One megaword (1 word = 4 bytes)
- 4 • One gigabyte

12-E.3 Your new laptop

The computer catalog advertises a laptop with 1.3 GB of memory:

- 1 • How many bytes, exactly, does the memory contain?
- 2 • How many bits, exactly, does the memory contain?
- 3 • Assume you use the entire memory to represent a single variable. How many possible values can that variable have? You are not asked to write down the exact number (*Hint*: don't try unless you own a paper factory) but the best approximation you can of the form 10^n for some n .
- 4 • If you *did* want to write the number on paper, 100 digits per line and 60 lines per page, how many pages would you need?

12-E.4 Size and transmission speed

You must transmit 128MB of data using a 128-Mb modem working at full capacity. How much time (within one second) will it take?

12-E.5 Octal arithmetic

- 1 • What is the octal representation of the decimal number *300,000*?
- 2 • What decimal number does the octal number *74,223* represent?
- 3 • What is, in both octal and decimal, the sum of the two octal numbers and *277,091*?

12-E.6 Hexadecimal arithmetic

13

Describing syntax

With the study of control structures and assignment we have started to encounter language constructs with an elaborate syntax structure, which programmers can nest within one another. Other syntactically interesting concepts will follow.

To reason about such constructs we need a better way of specifying syntax. Informal descriptions as in the presentation of control structures — “A **Conditional** consists of the keyword **if** followed by...” — are useful as explanations but not good enough as specifications.

← For example “[Syntax: Conditional](#)”, [page 179](#).

BNF (Backus-Naur Form) is available for that purpose. We’ll complement its study by an examination of related techniques for describing *abstract* syntax.

13.1 THE ROLE OF BNF

BNF lets us describe the *syntax* of programs and other texts with a precisely set structure. We’ve seen that the full description of a language must also cover *lexical* and *semantic* properties; for these we’ll need other tools.

← Chapter 3.

Before proceeding, make sure you the basic syntax concepts introduced in that earlier discussion are fresh in your mind: construct, terminal, nonterminal, specimen, syntax tree.

Touch of history: The original BNF

The history of programming languages starts in the nineteen-fifties. The first to achieve widespread recognition was FORTRAN, intended for scientific computation and designed by a team led by John Backus at IBM in 1954, with the compiler shipping in 1956. This success sparked the design of many new programming languages.



Backus

Soon American and European groups joined forces to design an international standard language which became known in 1958 as Algol 58. (The name stands for *ALGO*rithmic Language, and was ALGOL in upper case.) Its most influential version was the following revision, **Algol 60**.

The preparation of the Algol 60 specification revealed the need for better ways of describing syntax than the largely informal techniques used until then. John Backus, by then a member of the Algol committee, proposed a notation for describing the language, which became known as Backus Normal Form, the original BNF.

A 1964 letter to the *Communications of the ACM* from Donald Knuth (a professor of computer science at Stanford) suggested acknowledging the contributions of another committee member, Peter Naur from Denmark, by retaining the acronym but making it stand for “Backus-Naur Form”.

Many variants of BNF have been proposed since then. In the specification of his Pascal programming language (a descendant of Algol, first published in 1960) Niklaus Wirth from ETH Zurich used a graphical variant which has also been widely used.



Naur

Languages and their grammars

For our purposes a language is simply a set of “phrases”, where each phrase is a finite sequence of **tokens** from a certain “vocabulary”. For example in the Eiffel language a phrase is a class text, such as

```
class A end
```

the simplest one we can produce, made of just three tokens (two keywords and an identifier). The phrases encountered in practice — texts of useful classes — have many more tokens.

Not every sequence of tokens from the language’s vocabulary is a phrase of the language: **end A class** and **class class class** are not class texts. To define the syntax of a language is to specify which token sequences are phrases, and which are not. Such a specification is called a **grammar**:

Grammar

A grammar for a language is a finite set of rules for producing sequences of tokens from the language’s vocabulary, such that:

- 1 • Any sequence obtained by a finite number of applications of rules from the grammar is a phrase of the language.
- 2 • Any phrase of the language can be obtained by a finite number of applications of rules from the grammar.

← This is a more detailed version of the original definition on page [44](#).

So by applying the rules we get all of the desired language (clause [2](#) of this definition) and only the desired language (clause [1](#)).

Most languages of interest are potentially infinite; for example there's an infinite set of possible Eiffel class texts. But that theoretical possibility doesn't cause any practical problem, first because in our lifetime we'll deal only with a finite set of programs, but more importantly because every *phrase* — here every class text — is itself a **finite sequence** of terminals. The sequence might be very long, but it can't be infinite.

With a finite set of rules and an infinite language, you would have to keep applying the rules forever to produce all possible phrases, for example all Eiffel class texts. That again doesn't bother us: we don't need all classes, we only need those of interest to us — once we know that the rules are capable in principle of describing every possible class.

BNF is a notation for defining grammars. It's an example of a **metalanguage**: a *language* serving to describe other *languages*, such as programming languages.

BNF and the other techniques of this chapter apply not only to programming languages but to all *formal language*: artificial notations with a rigorously defined structure. HTML, the basic format of Web pages, and XML, a general-purpose format for structured data, are examples of formal languages that are not programming languages in the usual sense. In fact the [original research](#) was directed at understanding *natural* languages — whose complexity and irregularity exceed, however, the modeling power of BNF.

→ See at the end of this chapter: [“Touch of history: Classes of languages and grammars”](#), page 349.

BNF basics

To describe grammars we'll use a form of BNF called BNF-E, which serves in particular for the standard description of Eiffel. There are many other variants, such as Extended BNF (EBNF) defined by the International Standards Organization. “BNF” in the rest of this discussion means any BNF variant; any property specific to BNF-E is signaled as such. The differences are matters of style rather than substance.

BNF enables us to define a grammar for a language. A grammar, not *the* grammar, since different grammars may yield the same language.

A BNF grammar consists of the following parts, each a finite set:

- A finite set of **delimiters**; as [we've seen](#) these are the basic, fixed tokens of the language's vocabulary, such as keywords (**class**, **if** ...) and special symbols (period, colon, ...).

← [“TOKENS AND THE LEXICAL STRUCTURE”](#), page 47.

- A finite set of **constructs** representing structures of the language. Examples include **Class**, representing class texts, and **Conditional**, representing conditional instructions. The BNF-E convention is to start construct names with an upper-case letter and write them in **Green**. As you will remember, a particular instance of a construct is called a **specimen** of the construct; for example any conditional instruction is a specimen of **Conditional**. ← *Page 44.*
- A finite set of **productions**, each associated with a particular construct and specifying the form of its specimens. For example a production for **Conditional** defines the form of any conditional instruction: first the keyword **if**, then a specimen of **Boolean_expression** and so on.

Each production defines the syntax of specimens of a particular construct, in terms of other constructs and delimiters. Here for example is the production for **Conditional**:

Conditional \triangleq **if** Then_part_list [Else_part] **end**

This says that any specimen of **Conditional** — any conditional instruction — consists of the keyword **if**, a delimiter, followed by a specimen of the construct **Then_part_list**, followed optionally (the brackets signal an optional component) by a specimen of the construct **Else_part**, followed by the keyword **end**. The constructs **Then_part_list** and **Else_part** have their own productions.

Every production defines a single construct, here **Conditional**, appearing to the left of the symbol \triangleq , read “**is defined as**”; the BNF expression on the right specifies the structure of the construct’s specimen. This use of productions to define some constructs enables us to distinguish between the two kinds of construct:

- A construct that is defined by a production of the grammar is a **nonterminal** construct.
- Other constructs are **terminals**, for example (in the Eiffel grammar) **Identifier**, whose specimens are identifiers such as **PREVIEW**, and **Integer**, whose specimens are natural integers such as **34**. Since the grammar doesn’t provide a definition for a terminal construct, we must look elsewhere to know its syntax. The description will be provided at the **lexical** level.

The notions of terminal and nonterminal construct are not new; we saw them earlier in relation to abstract syntax trees, where terminals represent leaves and nonterminal represent internal nodes.

← “*Levels of language description*”, page 48.

← “*ABSTRACT SYNTAX TREES*”, page 45.

The reason for treating certain constructs as terminals and defining their properties outside of BNF is pragmatic: these constructs have a simple structure for which the full power of BNF (aimed at the description of potentially nested and complex structures such as those of classes and instructions) would be overkill. For example, an identifier is simply a sequence

of one or more characters, of which the first has to be a letter and any subsequent ones are letters, digits or underscores. This can be expressed easily by lexical techniques studied in a later [section](#) of this chapter.

In the BNF grammar we simply take the terminal constructs for granted.

The grammar as a whole defines the form that specimens of any construct may take, in the end in terms of terminals alone. Of particular interest will be the nonterminals describing the top-level structures of a language, such as **Class** in Eiffel; we'll call them **top constructs**. The *phrases* of the language — class texts in this example — are the specimens of the top construct.

→ [“THE LEXICAL LEVEL AND REGULAR AUTOMATA”](#), 13.6, page 343.

Distinguishing language from metalanguage

The production for **Conditional** illustrates that a BNF grammar includes three kinds of symbol:

- **Language** elements denoting delimiters of the language being described, for example — if the language is Eiffel — the keywords **if** and **end** in a production for **Conditional**, and special symbols such as **:=** in a production for **Assignment**.
- **Metalanguage** symbols: those of BNF itself, serving to express the productions. In the **Conditional** example they are \triangleq and the brackets [] enclosing an optional part.
- Names of **constructs**, both terminal and nonterminal, which belong to the metalanguage where they denote elements from the programming language. Terminals directly denote tokens, such as identifiers and integers; non-terminals denote sub-structures (for example every specimen of **Conditional** includes a substructure that is a specimen of **Then_part_list**).

Because of this mix of symbols from language and metalanguage we must be careful to avoid confusion. The rules are the following:

- Metalanguage elements (the symbols of BNF-E itself) appear in black.
- Names of constructs, such as **Conditional** (nonterminal) and **Identifier** (terminal), appear in **green**.
- Special symbols appear — like all programming language elements in this book — in **blue**. But this is not quite enough for symbols like brackets which would be easy to mistake for a metalanguage symbol. So they will be enclosed in straightquotes: for example **:"** denotes a colon as it will appear in the Eiffel text; and a production for any Eiffel construct that uses an opening bracket will denote it as **"[** to avoid any confusion with the metalanguage bracket signaling an optional part.
- For the other kind of delimiter, keywords such as **if** and **then**, we don't need quotes because the keywords are always written in **boldface blue**, avoiding any confusion. So we just let the keywords stand for themselves.

The term *specimen*, which may have surprised you the first time around, is similarly intended to avoid confusion. A specimen of a construct is a language structure that satisfies the properties of the construct, for example a particular Conditional instruction. The word “instance” would capture this notion, but it is already used to denote run-time objects corresponding to a class. An instance of a class is not the same thing as an instance of the construct **Class!** (One is a run-time object, the other a program text.) Using “specimen” for constructs removes any ambiguity.

← “[GRAMMAR. CONSTRUCTS AND SPECIMENS](#)”, page 43.

13.2 PRODUCTIONS

A production defines the syntax of specimens of one construct. It is of the form

$$\text{Construct} \triangleq \text{Definition}$$

where the left-hand side **Construct** states the construct being defined, and *Definition* specifies the syntax, in terms of constructs — terminals and nonterminals — and delimiters. Depending on the form of the *Definition* there are three kinds of production: Concatenation, Choice, Repetition.

Concatenation

A **Concatenation** production lists zero or more constructs in a certain order, some possibly enclosed in brackets [...] and then said to be **optional**. Our first production, **Conditional**, was an example::

$$\text{Conditional} \triangleq \text{if Then_part_list [Else_part] end}$$

Such a production specifies that every specimen of the new construct consists of a sequence (“concatenation”) of specimens of each the constructs listed, in the order given, except that the specimens of any of the optional constructs may be missing. In the example every specimen of **Conditional** consists of the concatenation of the keyword **if** (a terminal), a specimen of **Then_part_list**, optionally a specimen of **Else_part**, and the keyword **end**.

“Concatenation” simply means the linking of two or more elements as in a chain — *catena* in Latin. The word is often used in a programming: to *concatenate* two character strings is to join them into a single string. Its use for BNF is a bit pretentious, as we could talk of “*sequence* productions”. But in the programming language we also have *sequences* of instructions, our first control structure. Again to avoid confusion between language and metalanguage, we use “Sequence” for the Eiffel construct and “Concatenation” for BNF productions. In a similar way the Choice productions evoke conditionals, and the Repetition productions evoke loops, but the terminology is distinct to avoid confusion. The analogies are, however, interesting, and will be explored further below.

→ “[TURNING A GRAMMAR INTO A PARSER](#)”, 13.5, page 342.

Choice

A **Choice** production lists one or more constructs, separated by vertical bars |. An example is the production defining instructions::

$$\text{Instruction} \triangleq \text{Conditional} \mid \text{Loop} \mid \text{Compound} \mid \text{Assignment} \mid \text{Call}$$

A Choice production specifies that every specimen of the new construct consists of exactly one specimen of one of the constructs listed. (For that reason, the order of their listing is, in this case, irrelevant.) In the example a specimen of **Instruction** is a specimen of either **Conditional**, or **Loop** etc. In ordinary language, we would say “An instruction is one of: a conditional, a loop, a compound, an assignment or a call”.

→ This is only an example, omitting a few of the kinds of instructions available in Eiffel.

We may indeed from now on say “An **X**”, for some construct name **X**, as an abbreviation for “A specimen of **X**”; for example: “A **Conditional**”.

Repetition

Finally, a **Repetition** production lists two constructs, one a nonterminal to be repeated, and the other, usually a terminal, serving as separator. For example we may specify compound instructions (instruction sequences) as

$$\text{Compound} \triangleq \{ \text{Instruction} \text{ ";" } \dots \}^*$$

← “SEQUENCE (COMPOUND INSTRUCTION)”, 7.4, page 145.

meaning: a specimen of **Compound** is made of a succession of zero or more specimens of **Instruction**, each separated from the next, if any, by a semicolon. According to this rule, possible specimens of **Compound** are of the forms:

- Nothing at all (Repetition of zero **Instruction** specimens)
- *inst1*
- *inst1 ; inst2*
- *inst1 ; inst2 ; inst3*
- etc.

where *inst1*, *inst2*, *inst3* ... are instructions.

We saw that the semicolon is optional. Although this property can be expressed through the grammar, it’s more convenient to use the above production and add a separate non-BNF tolerance rule stating that a missing semicolon is harmless.

The asterisk — a well-established symbol from the mathematical theory of formal languages — means “zero or more”; the three dots suggest repetition; the braces { } are just for grouping.

With this production, the repetition may be empty; the syntax indeed allows for an empty **Compound**. This is convenient for such cases as

<pre>if <i>some_condition</i> then else <i>instruction_1</i> <i>instruction_2</i> end</pre>	[S1]
---	------

where the empty **then** part is legal since syntactically it's a just an empty **Compound**. (In terms of programming style this is not a tidy structure and if it is to persist you should clean it up, for example to

<pre>if not <i>some_condition</i> then <i>instruction_1</i> <i>instruction_2</i> end</pre>	[12]
--	------

but the first form [11] with an empty **Compound** can be useful when you are changing your software, moving instructions around, and a **Compound** like the **then** part of [11], previously containing instructions, temporarily finds itself empty.)

For some constructs an empty repetition is not desirable. Then you'll use a variant of the Repetition that instead of the asterisk * uses a ⁺, also a standard symbol from mathematical language theory, meaning "one or more". Here for example is the production for **Then_part_list**, given with the other constructs related to conditional instructions (which we can now see in full since all types of production have been introduced):

<pre>Conditional \triangleq if Then_part_list [Else_part] end Then_part_list \triangleq {Then_part elseif ... }⁺ Then_part \triangleq Boolean_expression then Compound Else_part \triangleq else Compound</pre>

The Repetition production for `Then_part_list` indicates that a specimen of this construct — a more complete name would be “then and possibly elseif part list” — is of one of the forms

- `cond1 then inst1`
-- One specimen of `Then_part`
- `cond1 then inst1 elseif cond2 then inst2`
-- Two specimens of `Then_part`
- `cond1 then inst1 elseif cond2 then inst2 elseif cond3 then inst3`
-- Three specimens of `Then_part`

and so on, for boolean expressions `cond1`, ... and instructions `inst1`, ... Note that the `Then_part_list` is not optional in the Concatenation production for `Conditional`, so there will always be at least one `Then_part`, of the form `some_condition then some_compound`; if there's more than one they will be separated by `elseif` as shown.

Rules on grammars

In BNF — any variant — an obvious rule on productions is that every component appearing on the right-hand side (the *Definition* of a construct) must be one of: a delimiter; a terminal construct; a nonterminal construct.

This corresponds to the three sets that officially make up a BNF: delimiters, terminals, nonterminals. To define a grammar in practice it suffices to list the productions, which yield these three sets through simple conventions:

- 1 • **Delimiters** are self-describing, with the conventions defined: keywords **stand out**, special symbols appear "in quotes".
- 2 • Any other `identifier` appearing in a production denotes a construct.
- 3 • If such a construct appears on the left-hand side of at least one production, it is a **nonterminal** (since the production defines a structure for its specimens, in terms of other elements).
- 4 • Otherwise the construct is a **terminal**. In that case its definition must appear in the lexical part of the language specification.

Case 3 assumes that a given nonterminal may appear on the left of more than one production. This is permitted by most BNF variants other than BNF-E, with the convention that two separate productions for the same construct

- $A \triangleq Def1$
- $A \triangleq Def2$

← “BNF basics”,
page 329.

← “Distinguishing lan-
guage from metalan-
guage”, page 331.

are to be interpreted as

$$A \triangleq \text{Def1} \mid \text{Def2}$$

Alternatively or in addition, such BNF variants allow mixing the various production mechanisms — Concatenation, Choice, Repetition — in a single production, as in

$$A \triangleq B \mid C [D] \{E ";" \dots\}^*$$

WARNING: Not permitted. See next..

BNF-E disallows such mixing of production styles:

Touch of Methodology: BNF-E rules

- Every nonterminal must appear on the left-hand side of exactly one production, called its “**defining production**.”
- Every production must be of one kind: Concatenation, Choice or Repetition, following the rules defined above.

So for the example given you must use three productions:

$$\begin{aligned} A &\triangleq B \mid \text{Concat} \\ \text{Concat} &\triangleq C [D] \text{Repet} \\ \text{Repet} &\triangleq \{E ";" \dots\}^* \end{aligned}$$

The correct form.

Along with a few notational conventions, this is the major specificity of BNF-E among BNF variants.

In writing language definitions I have found that while this rule introduces more nonterminal constructs — such as **Concat** and **Repet** here, and **Then_part_list** in the earlier example — it yields simpler, more understandable and more effective language descriptions.

It also leads to a better assessment of **language size**. If you can stuff different mechanisms into a single production, you may give the impression of a small language whereas it’s actually quite complex. Since you can’t do that with BNF-E, the number of productions is a good indicator of the actual syntactical complexity, as the extra nonterminals do represent real concepts. In other words, the notation keeps the language designer honest.

13.3 USING BNF

We have now seen all of BNF. The following pragmatic observations will help you apply the techniques effectively.

Applications of BNF

BNF descriptions enable you to:

- Understand the syntax description of existing languages, in particular (but not only) programming languages.
- Define the syntax of languages you need to design.
- Write syntax analyzers, or *parsers*.

The second application is not as far-fetched as it sounds. Although you may not have to design a general-purpose programming language — a competitor to C, Java or Eiffel — as part of your first job, programmers have to define “*little languages*” all the time. Whenever you write a program that will process some data, the format of the data is a language, and if that format isn’t trivial BNF is often the right way to define it. Exercises in this chapter ask you to write the BNF specifications of a few such examples.

The third application, parsing, is useful for writing writing **compilers** and other tools that must process languages. One of the first tasks of such a tool is to reconstruct the structure of the text — in the form of an **abstract syntax tree** — from the external appearance of the text. That’s what the parser does, as studied in the [chapter](#) on programming tools. Any parser needs a formal description of the language it’s supposed to parse; this will come from a grammar in BNF.

→ “[Parsing](#)”, page 356.

Language generated by a grammar

We may see a BNF grammar in two complementary ways, following from the two clauses in the [definition](#) of the notion of grammar:

← “[Grammar](#)”, page 328.

- It is a *recognition* mechanism: to determine whether a certain sequence (and terminal specimens and delimiters) is a phrase of the language, and if so to reconstruct its syntactic structure. This is the view of interest when you are, for example, writing a parser.
- The grammar is also a *generation* mechanism: by applying its rules you may produce, one after the other, all the phrases of the language.

The second view is less often useful in practice but important all the same. Let’s explore it a bit further. To produce all the possible specimens of any nonterminal — in particular the top symbol — it suffices to look up the production defining it (remember that in BNF-E there’s only one):

- P1 • For a Concatenation, the specimens are all possible sequences of specimens of the constructs listed, including those where optional ones are ignored.
- P2 • For a Repetition, the specimens are all sequences of zero or more (one or more in the case of “+”) specimens of the construct listed, with the given separator in-between.
- P3 • If at any step of the previous steps you encounter a nonterminal, apply the same process to it so as to get its own specimens.
- P4 • For a Choice, apply the previous steps to all the the constructs listed, and collect all the specimens that you get from any of them.

These four phrase-generation mechanisms, carried out as long as at least one of them is applicable, will eventually yield all the phrases of the language. This is a potentially infinite process, as indeed we have seen that most languages of interest are infinite.

In carrying out this process for a nonterminal **A** whose production uses **B**, you may have to apply the same rules — in steps **P3** and **P4** — to other constructs.

Recursive grammars

The last observation may raise some alarm. What if, applying the process to **A**, we have to apply to another construct **B** and in so doing we encounter **A** again? We must make sure that this will terminate.

The production of **Compound** provides a good example. It reads:

$$\text{Compound} \triangleq \{\text{Instruction} \text{ ";" } \dots\}^*$$

involving the construct **Instruction**; but that construct is itself defined as

$$\text{Instruction} \triangleq \text{Conditional} \mid \text{Compound} \mid \dots \text{Other choices} \dots$$

involving **Compound**, as well as **Conditional** whose definition also uses **Compound**. If we try to understand **Compound**, by looking for its specimens according to the rules given above, we'll need to determine the specimens of **Instruction**; but then this will require us — by the same rules — to look for specimens of **Compound**. This seems like circular and meaningless reasoning.

Such a definition, appearing to define a concept in terms of itself (directly or, as here, indirectly) is said to be **recursive**. Recursion — the use of recursive definitions — pops its head in almost all areas of programming and we'll have an entire [chapter](#) devoted to it. But since there is almost no useful grammar without recursion we should already convince ourselves that such grammars can actually make perfect sense.

→ [Chapter 16](#).

Let's see the concepts on a smaller example. Consider a mini-language with three keywords **heads**, **tails** and **stop**, no other terminals, and **Game** as its top construct defined by the following grammar involving one Choice and two Concatenation productions:

$$\begin{aligned} \text{Game} &\triangleq \text{Head_start} \mid \text{Tail_start} \mid \text{stop} \\ \text{Head_start} &\triangleq \text{heads Game} \\ \text{Tail_start} &\triangleq \text{tails Game} \end{aligned}$$

This is similar to the situation with **Compound**, **Instruction** and **Conditional**: three nonterminal constructs defined in terms of each other.

Quiz time!

Can you find examples of specimens of **Game** in the language defined by the above grammar? What specimens do **Head_start** and **Tail_start** have?

(The answer follows.)

Because of the recursion the grammar might seem meaningless. But let's take a pragmatic view by asking if we can use the grammar, through the construction process described above, to **generate** specimens. Notice that in the production for **Game** one of the branches, **stop**, is a terminal. So we do have a first phrase (specimen of **Game**):

- **stop**

But now since both **Head_start** and **Tail_start** are defined in terms of **Game**, we can use the information just gained about **Game** to get a specimen of each of these constructs: the productions tell us that **heads stop** is a specimen of the first, and **tails stop** of the second. Now we bring this information back into the production for **Game**, which has these two constructs among its choices, to get two new specimens of **Game**:

- **heads stop**
- **tails stop**

Applying the same process again, on to **Head_start** and **Tail_start** and back to **Game**, gives us four more specimens:

- **heads heads stop**
- **heads tail stop**
- **tails heads stop**
- **tails tails stop**

And so on. At this point we see the pattern: a specimen of **Game** is any sequence of one or more occurrences of **heads** and **tails** (arbitrarily intermixed), followed by **stop**. More precisely: from what we've seen it's easy to prove that any such sequence is a specimen; a slightly more delicate question is whether, conversely, these are the only possible specimens.

→ Exercise [13-E.3](#),
[page 351](#).

The very simple language defined by this grammar for the top construct **Game** might represent all the possible coin-tossing sequences by a player who at some point gives up, crying “**stop!**”. The non-recursive grammar

$\begin{aligned} \text{Game1} &\triangleq \text{Throw_sequence stop} \\ \text{Throw_sequence} &\triangleq \{\text{Throw } \dots\}^+ \\ \text{Throw} &\triangleq \text{heads} \mid \text{tails} \end{aligned}$

generates the same language through three productions, one each of Concatenation, Repetition (with empty separator) and Choice.

This example illustrates the earlier remark that it makes no sense to talk of *the* grammar for a language, since any non-trivial language, even one as simple as this example, can be generated by many possible grammars. The other way around, of course, a grammar defines just one language.

In applying the production rules to generate the language, we have used a strategy that favored terminals (here delimiters **stop**, **heads** and **tails**, but other terminals would play the same role) over nonterminals. By choosing a different policy, we would get into an infinite cycle without ever producing a phrase: start by choosing the first possibility, **Head_start** for **Game**; for **Head_start**, we get **heads Game**; for **Game**, choose again **Head_start**; and so on forever, without ever generating a phrase. To avoid such situations, a language generation strategy needs strategies, or *heuristics*; a possible one is, for a Choice, always to start (at step **P4**) with a production using terminals only, if there's one, and otherwise with a production that starts with a token.

→ On this notion see
also “[Interpretation vs. compilation](#)”, [page 446](#).

Even with such heuristics the process will not yield anything if a grammar is *entirely* recursive. It needs at least some token choices to get the process started. Grammars such as

$A \triangleq A$

or

$$A \triangleq B$$

$$B \triangleq A$$

are useless. These issues will be discussed further in the [chapter](#) on recursion. → [Chapter 16](#).

A more delicate case is a grammar that does have tokens but is *left-recursive*, as in

$$\text{Instruction} \triangleq \text{Compound} \mid \text{Assignment}$$

$$\text{Compound} \triangleq \text{Instruction} ";" \text{Compound}$$

where for simplicity we take **Assignment** as a terminal (meaning we accept it's defined somewhere else). This grammar is meaningful, since it permits instructions of the form

- *assignment_1*
- *assignment_1 ; assignment_2*

and so on. To obtain a constructive view of such recursive definitions, we need a general theory of recursive definitions, [sketched](#) later.

→ [“MAKING SENSE OF RECURSION”](#), [16.5, page 384](#).

One form of grammars that avoids these complications and makes it possible to write simple parsers is known as **LL (1)**, characterized by the property that the first terminal starting a phrase is enough to choose between variants of any nonterminal. Eiffel is close to LL (1): restricting ourselves to instructions, we know that we have a **Conditional** if the first token is **if**, a **Loop** if it is **from** and so on.

13.4 DESCRIBING ABSTRACT SYNTAX

The syntax that we have studied in this chapter is the **concrete** syntax of a program: it describes the full structure of program texts, including keywords — **if**, **do**, **class** ... — and other delimiters that serve a purely syntactic role: they avoid syntactic ambiguity but do not carry any semantics of their own.

[Earlier](#) we have encountered **abstract** syntax, which discards these elements and retains the structurally meaningful ones only.

← [“ABSTRACT SYNTAX TREES”](#), [page 45](#).
Original figure on page 46.

We saw how to describe the resulting syntactic structures through **abstract syntax trees**, such as the one representing our example *PREVIEW1* class, reproduced on the right.

As noted then, it's easy to deduce a notion of *concrete* syntax tree, which returns all the symbols of the input text. Some compilers indeed construct such a tree, but this is usually not necessary: in subsequent phases of compiling, such as semantic analysis, code generation and optimization, the syntactic markers don't play any role; all that's needed is a representation of the program's structure, which the abstract syntax tree precisely provides.

If we want directly to describe the abstract syntax of a language, without going through concrete syntax, we don't need a new formalism. It suffices to use BNF, omitting all tokens that are not constructs of the lexical grammar, in particular keywords; the right-hand side of the last production shown, for *Compound*, would now be just *Instruction Compound*. Such a grammar is not useful for applications such as parsing and compiling input texts — which obviously require a concrete grammar as those discussed elsewhere in this chapter — but it is appropriate to capture the structural properties of texts, unaffected by details of their physical appearance.

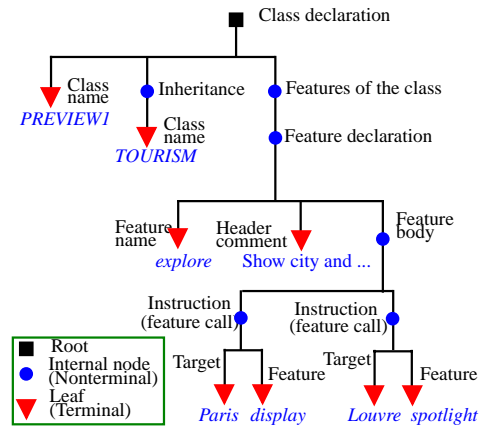


Figure from page 46.

13.5 TURNING A GRAMMAR INTO A PARSER

One of the applications of BNF is, as we have have noted, to guide the construction of compilers, starting with the *parsing* phase. Compilers are usually **syntax-driven**: the parsing phase constructs an abstract syntax tree, and successive compilation phases continue to work on this data structure, repeatedly adding semantic information (this is known as *decorating* the tree).

A detailed discussion of how to build such a syntax-driven compiler, or just a parser, is beyond our scope here, but to get an idea of possible techniques you may wish to look at the [EiffelParse](http://archive.eiffel.com/products/parse.html) library. EiffelParse is not the most efficient parsing mechanism available (for a widely used Eiffel parsing tool based on more traditional techniques, look up the “Gobo” library), but it provides a clear, practical illustration of how to apply the object-oriented principles of this book in full to parsing and compiling.

archive.eiffel.com/products/parse.html

The idea behind EiffelParse is to turn a BNF-E grammar directly into a set of classes. For each construct of the grammar, you will write a small class that inherits from one of the EiffelParse classes *AGGREGATE* (for concatenation productions), *CHOICE* and *REPETITION*. In the *AGGREGATE* case, for example, the class will simply list the various

components of the production's right-hand side, each associated with a class similarly describing a construct. You have to be a bit careful about left recursion, but otherwise the classes are a mirror image of the BNF-E productions; in fact a translator, YOOC, from Monash University, can produce the classes directly from the grammar.

To parse an input text, it then suffices to call the EiffelParse procedure *parse* on the construct of interest. This produces an abstract syntax tree. You can then add *semantic* processing of any kind through procedures of the syntax classes. The process shows the power and elegance of object-oriented modeling for language processing.

13.6 THE LEXICAL LEVEL AND REGULAR AUTOMATA

For terminal constructs such as **Identifier** and **Integer**, the BNF grammar does not provide a production, leaving the specification instead to the lexical level. For that reason the terminal constructs are also called **lexical constructs**. Their specification appears in a “lexical grammar” complementing the BNF grammar.

Lexical constructs in BNF

At the syntax level and hence in BNF, tokens were terminals: atoms with no further structure of interest. At the lexical level, we become interested in their internal makeup. For example (using Eiffel conventions):

- An **Identifier** is a sequence of one or more characters, of which the first is a letter (upper or lower case) and any subsequent one is a letter, digit or underscore “_”.
- An **Integer** is a sequence of one or more decimal digits (0 to 9), which may also contain underscore characters to separate groups of digits, usually three by three, in long numbers: `123_456_789`.
- An **Integer_constant** is an **Integer** optionally preceded by a sign, + or -.

It is in fact possible — as an *exercise* invites you to check for yourself — to specify such terminal constructs in BNF. That is not, however, the usual practice. For such simple constructs, language definitions generally take advantage of specific *lexical* techniques, which we'll now study. This avoids loading the grammar with productions for basic structures that can be described more simply, and reserves the BNF grammar for specifying the higher-level structures of the language, in particular those permitting an arbitrary level of nesting.

→ “BNF for lexical grammars”, 13-E.2, page 351.

Correspondingly, compilers don't use the parser to decode specimens of terminals, but a simpler tool known as a *lexical analyzer*.

Regular grammars

We may express the structure of lexical constructs such as the examples above through a **regular grammar**, a toned-down version of BNF.

The non-terminals of such a grammar are constructs such as **Identifier** and **Integer_constant**, which will be used as terminals in the BNF. To express their structure the regular grammar has its own lower-level terminals, usually character categories such as

$\text{Letter} \triangleq 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' $ $'n' 'o' 'p' 'q' 'r' 's' 't' 'u' 'v' 'w' 'x' 'y' 'z'$ $\text{Decimal_digit} \triangleq '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'$ $\text{Underscore} \triangleq '_'$
--

Letter and Decimal_digit have a simpler definition, see below.

each expressed as a Choice between **single characters**, shown in quotes. Such constructs are *really* terminal (atomic): we can't decompose them any further.

It is common to provide a special notation for consecutive characters, so that we may rewrite the production for **Letter** as

$\text{Letter} \triangleq 'a' .. 'z' 'A' .. 'Z'$
--

using this opportunity to add the upper-case variant, and similarly define **Decimal_digit** as $'0' .. '9'$

A regular grammar may have the same kinds of production as in BNF, but with slightly different conventions and significant restrictions:

- You can use a **Choice** as just seen, possibly with character intervals.
- You may define a lexical construct by **Concatenation**, but this doesn't assume breaks (spaces, new lines etc.) between the concatenated elements. If you define a construct as **A B**, any of its specimens will be made of a specimen of **A** followed by a specimen of **B** with nothing in-between. (If you want to introduce breaks you can define this notion in the regular grammar.)
- A **Repetition** will take a simpler form: just **A*** or **A⁺** where **A** is a previously defined construct, with no notion of delimiter. These two forms denote “zero or more specimens of **A**” and “one or more specimens of **A**”, with no notion of delimiter, and again no intervening break.
- **No recursion** is permitted in the grammar: you can't have the definition of **A** use **B** and conversely (including when they are the same). A simple way to ensure this is to make the *order* of the rules significant — in BNF it's not — and add the rule that a construct definition may only use previously defined constructs.

Unlike BNF-E, a regular grammar allows mixing the different kinds of production (since the formalism is much more restricted).

← "[Touch of Methodology: BNF-E rules](#)", page 336.

With such a regular grammar we can give a precise definition of the lexical notions of identifier and integer constant:

$$\text{Identifier} \triangleq \text{Letter} (\text{Letter} \mid \text{Digit} \mid \text{Underscore})^*$$

$$\text{Integer_constant} \triangleq \text{Decimal_digit}^+$$

The expressions permitted by the rules just defined are called **regular expressions**. A language that can be described by a construct of a regular grammar is a **regular language**. We may note the following property:

Theorem:
Canonical form of a regular language

Any regular language can be described by a regular grammar whose production right-hand sides do not include any non-terminal.

Proof: this is a simple consequence of the prohibition of recursive definitions. Starting from a regular grammar, order the productions, as discussed above, so that any non-terminal appearing on the right-hand side of any production has been defined by a previous production. Then for each production in turn, if the right-hand side has a non-terminal N , replace it by the right-hand side for N . Since the same process has already been applied to N , we'll get terminals only.

For example with

$$\begin{aligned} A &\triangleq T1 \mid T2 \mid T3^* \\ B &\triangleq T4^+ \mid A \\ C &\triangleq A B \end{aligned}$$

this process yields the alternative — not necessarily clearer — definition

$$\begin{aligned} A &\triangleq T1 \mid T2 \mid T3^* && \text{-- No change} \\ B &\triangleq T4^+ \mid T1 \mid T2 \mid T3^* && \text{-- Obtained by replacing } A \\ C &\triangleq (T1 \mid T2 \mid T3^*) (T4^+ \mid T1 \mid T2 \mid T3^*) \end{aligned}$$

which generates the same language. Another way of stating the theorem is that any regular language can be described by a *single* regular expression (the right-hand side for C in the last production above).

The theorem illuminates the principal restriction of regular languages: they don't support recursive nesting. We saw that programming languages such as Eiffel have **Conditional** instructions that may contain other instructions of the same kind, or of different kinds such as **Loop** which in turn can contain conditionals, allowing nesting up to any desired depth. With BNF we can describe this finitely through recursively defined productions. With regular grammars we can't.

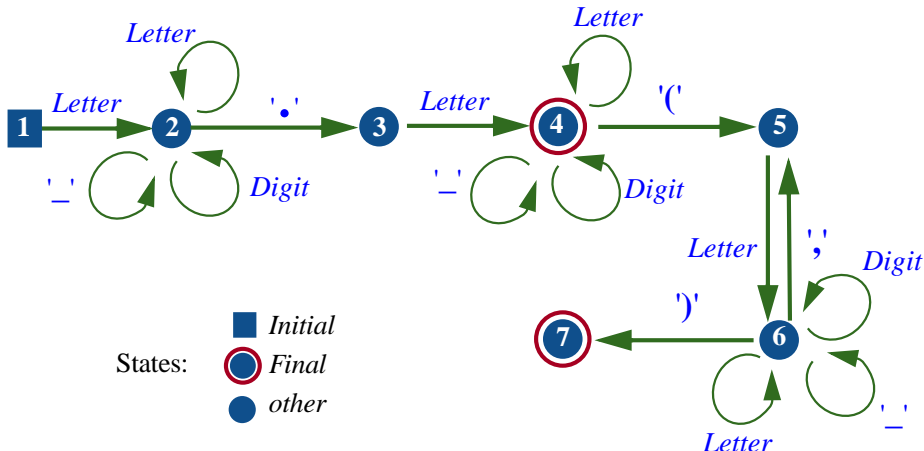
Regular grammars are well suited, however, for defining the usually simple tokens that make up the elementary fabric of programs. To express that a certain kind of token has specimens made up (say) of one or more character of a certain kind, followed by any of 3 specified characters, followed optionally by characters of another kind, regular expressions are just the ticket.

Finite automata

Behind regular expressions stands the mathematical theory of *finite automata*. Let's take a glimpse at it, if only for the visual illustration that it provides. (There's much more to the theory than this brief introduction.)

A finite automaton is a kind of graph with nodes representing states and edges labeled by elements of some basic set, here characters or character categories. The following example corresponds to the structure of qualified feature calls in Eiffel, possibly with arguments, as in [Line8.extend](#) (*new_station*):

Without spaces, for simplicity.



Finite automaton for recognizing feature calls

As the name suggests, we can view a finite automaton as a machine to process input strings. The automaton will start from the state marked **initial** and then, at each step, follow an edge, if any, labeled by the next symbol from the input. This is called a **transition**. For the input $x9.f_g(aa)$, the above automaton starts in state **1**; the input symbol x causes a transition to state **2**; then 9 causes a transition from state **2** to itself; the period takes us to state **3**, then f to **4**, where it stays for $_$ and g ; the argument list then causes successive transitions to **5**, **6**, **6** again for the second a , and the state **7**, marked as a **final** state.

The **language recognized by a finite automaton** is the set of strings which, as in this example, will take the automaton to a final state through a set of transitions. A string does *not* belong to that language if, when trying to apply this process to the string:

- Either the automaton reaches a state that has no transition matching the next input symbol (as with the input string $a.b.c$, where $a.b$ takes the automaton to state **4**, from which there's no edge labeled with a period).
- Or, having consumed all symbols from the input, it reaches a non-final state (as with the input string $a.b($, taking us to state **5** which is not final).

Actually legal Eiffel, but not included in this mini-language of feature calls where we only accept single-dot calls.

A basic theorem states that for any language described by a regular grammar is recognized by a finite automaton, and conversely. Without proving the theorem, we may illustrate it by noting that the language recognized by the above automaton is also the language generated by the last construct of the following regular grammar:

Identifier	\triangleq	Letter (Letter Digit Underscore)*
Another_argument	\triangleq	"," Identifier
Argument_list	\triangleq	"(" Identifier Another_argument* ")"
Feature_call	\triangleq	Identifier "." Identifier [Argument_list]

The feature calls recognized by this grammar are only a subset of those possible in Eiffel, where expressions, like instructions, can be nested: a feature argument can also be an expression, as in $x.f(y.h(z.i))$, which the above lexical grammar and the associated finite automaton can't handle since they limit any argument to a single identifier. As soon as you venture beyond tokens, you'll want the full power of BNF. (Note also how the convention for repetitions, with its provision, is more convenient in BNF-E, where the equivalent of the above **Argument_list** would be defined by a single production with the right-hand side $\{\text{Identifier } "," \dots\}^+$.)

The above automaton is *deterministic* in the following sense: it has only one initial state, and from any state there is at most one edge for any given character; as a result, the recognition process illustrated above can take, at any step, at most one transition. *Nondeterministic* finite automata don't have these properties. It turns out, however, that they do not change the class of languages that can be recognized.

Finite automata provide the basis for *lexical analyzers*, the part of compilers that takes care of recognizing tokens. It is indeed not hard to see how to define a finite automaton from a regular expression, and from that definition build a program that will recognize tokens through the process just illustrated.

Context-free properties

The theory of formal languages distinguishes a number of levels including, from simplest to more sophisticated:

- **Regular languages** — those which can be described by a regular grammar.
- **Context-free languages**, which can be described by a grammar made of production rules with possible recursion, as in BNF.
- **Context-sensitive languages**, for which such production rules are no longer sufficient.

As an example of why context-free languages are not enough, consider the *type rules* that govern many programming languages. Eiffel indeed requires that whenever you use an entity x , for example in an expression such as *some_function(x)* or an instruction such as *x.some_procedure*, there must have been, in an enclosing program unit — the enclosing routine, or the enclosing class — a declaration of the form

```
x: SOME_TYPE
```

specifying that x is a formal argument or local variable of the routine, or an query of the class. Otherwise, as you know, your program is illegal and compilers are required to reject it. But this is different from an error such as

```
if c then a + b end
```

which violates the BNF grammar (whose production specifies an **Instruction** after **then**, whereas $a + b$ is an **Expression**).

There are many more examples of construct specimens that conform to the BNF but are not acceptable, such as an instruction using a variable x that hasn't been declared, or has been declared with a type that doesn't permit what the instruction does with it.

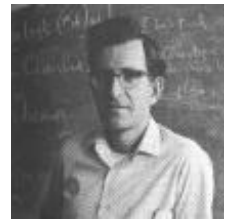
Context-free grammars and BNF can't capture such type rules; a grammar to handle them would have to be context-sensitive. Where BNF rules define a nonterminal A as a sequence γ of terminals and non-terminals, a context-sensitive grammar has rules allowing replacement of $\alpha A \beta$ by $\alpha \gamma \beta$; here α and β are the "context" around A .

In practice, no context-sensitive grammar formalism exists that matches the simplicity and practicality of BNF. Because programming languages need type rules and other context-sensitive properties, the solution adopted in practice by compilers is to:

- Rely on regular grammars to describe the tokens of the language, and build lexical analyzers on the basis of associated finite automata.
- Rely on BNF or equivalent to handle the context-free aspects of the language — the overall, usually nested, structure of programs — and on associated techniques for parsing that structure.
- Enforce all other checks — the context-sensitive aspects, such as type rules — through additional mechanisms, either based on formalisms for the description of context-sensitive aspects (such as “attribute grammars”) or programmed in an *ad hoc* way in the compiler.

Touch of history:
Classes of languages and grammars

The classification of languages into regular (*Type 3*), context-free (*Type 2*), context-sensitive (*Type 1*) and unrestricted (*Type 0*, recognizable by any general automaton or “Turing machine”), comes from articles published in 1956 and 1959 by Noam Chomsky, then as now a professor at MIT, and Marco Schützenberger from the University of Paris. Chomsky, also famous as a political activist, was interested in the structures of *human* languages, for which his work started a whole new school of linguistics; but it also proved seminal for the understanding of programming languages and other artificial notations.



Chomsky

13.7 FURTHER READING

Dick Grune, Henri E. Bal, Cerial J.H. Jacobs and Koen G. Langendoen: *Modern Compiler Design*; Wiley, 2000.

A good description of current compiler technology.

Steven S. Muchnick: *Advanced Compiler Design and Implementation*; Morgan Kaufmann, 1997.

Another recent text, up to date on many important compiler techniques.

Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman : *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

An older textbook on compilers, still considered standard.

13.8 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A formal language, such as a programming language, is a set of *phrases* built from a basic vocabulary according to precise rules.
- Most interesting formal languages are infinite.
- BNF is a formalism for describing a formal language from a finite set of rules called “productions”.
- Each production of a BNF grammar describes the structure of a certain construct, or “non-terminal”, from other non-terminals as well as atomic constructs or “terminals”
- A production defines a construct by one of: concatenation of other constructs, possibly with optional components; choice between other constructs; or repetition of another construct.
- Compilers and other language analysis tools use grammars for decoding, or “parsing”, the structure of input texts.
- BNF can also describe abstract syntax which (unlike “concrete” syntax) discards keywords and other elements that do not directly carry a semantic value of their own.
- For the elementary components of input texts, such as identifiers and constants, BNF is usually overkill; simpler descriptions can be obtained through *regular* grammars, where productions can’t be recursive and as a result do not support nesting. Regular expressions are closely associated with mathematical devices known as finite automata.
- BNF covers the class of “context-free” languages but does not capture “context-sensitive” aspects such as type rules.

New vocabulary

(Also remember, from the [presentation](#) of basic syntax concepts: Construct, Lexical, Nonterminal, Specimen, Syntax, Terminal.)

← “[NESTING AND THE SYNTAX STRUCTURE](#)”, page 44.

Choice production	BNF	Concatenation production
Defining production	Grammar	Lexical construct
Lexical grammar	Metalanguage	Phrase
Production	Recursive grammar	Repetition production
Top construct	Vocabulary	

13-E EXERCISES

13-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

13-E.2 BNF for lexical grammars

Write a BNF grammar that fully describes the Eiffel forms of **Identifier**, **Integer** and **Integer_constant**. ← *“Lexical constructs in BNF”, page 343.*

13-E.3 Language defined by a recursive grammar

Consider the language defined by the grammar with top construct **Game**: ← *In “Recursive grammars”, page 338.*

- 1 • Prove that any specimen of **Game1** in the non-recursive grammar, that is to say any sequence of one or more **heads** or **tails** followed by a single **stop**, is a specimen of **Game**.
- 2 • Conversely, is any specimen of **Game** a specimen of **Game1**? Prove your answer.

13-E.4 Regular grammar

Define the language generated by the **Game** construct through a single regular expression. ← *In “Recursive grammars”, page 338.*

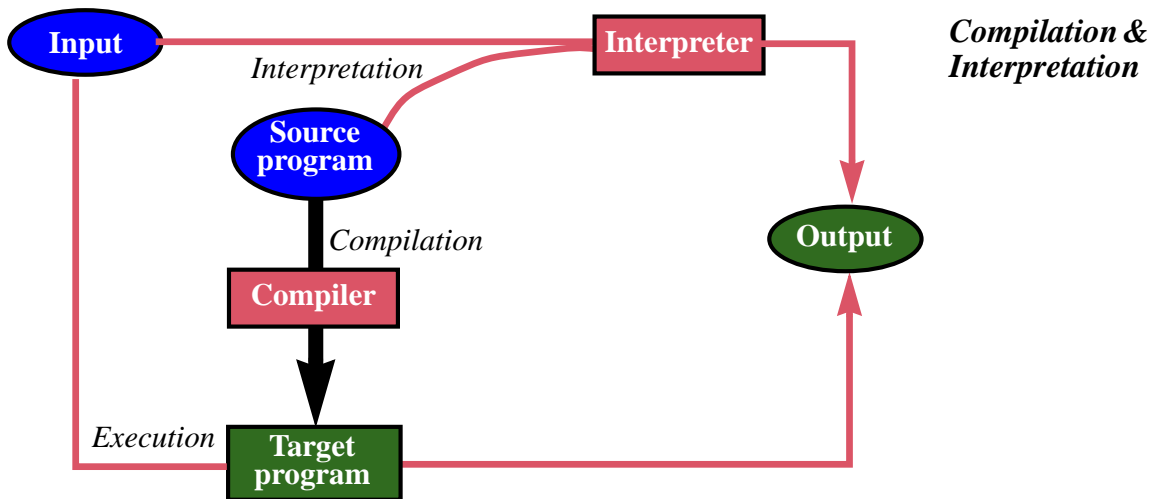
14

Programming languages

15

Compilers and friends: the basic software tools

15.1 COMPILATION VS INTERPRETATION



15.2 ESSENTIALS OF A COMPILER

Lexical analysis, regular languages and finite automata

Parsing

Semantic analysis, code generation and optimization

What else a compiler does

ww

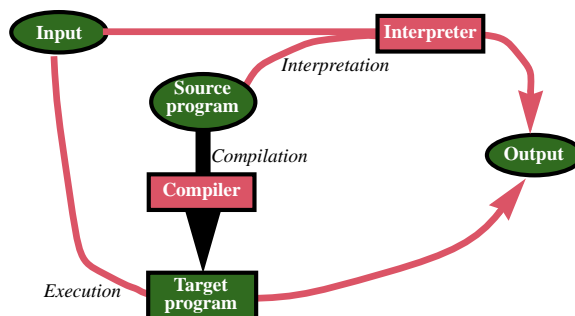
15.3 LOADING, LINKING AND ALL THAT

15.4 TEXT AND PROGRAM EDITORS

15.5 CONFIGURATION MANAGEMENT

15.6 VIRTUAL MACHINES

15.7 SOFTWARE DEVELOPMENT ENVIRONMENTS



PART III:

Algorithms and data structures

We

16

Recursion and trees



The cow shown laughing on the Laughing Cow® box holds, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably — I say “presumably” because here my eyesight fails me, I don’t know about yours — holding, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and

presumably... (by now you get the idea).

This 1921 advertising gimmick, still doing very well thank you, is an example of a structure defined *recursively*, in the following sense: www.bel-group.com.

Recursive definition

A definition for a concept is recursive if it involves one or more instances of the concept itself.

“*Recursion*” — the use of recursive definitions — has applications throughout programming: it yields elegant ways to define *syntax structures*; we’ll also see recursively defined *data structures* and *routines*.

We may say “*recursive*” as an abbreviation for “recursively defined”: recursive grammar, recursive data structure, recursive routine. But this is only a convention, because we can’t say that a concept or a structure is by itself recursive: all we know is that we can *describe* it recursively, according to the above definition. Any particular notion — even the infinite Laughing Cow structure — may have both recursive and non-recursive definitions.

When proving properties of recursively defined concepts we will use recursive *proofs*, which generalize inductive proofs as performed on integers.

Recursion is *direct* when the definition of A cites an instance of A ; it is *indirect* if for $1 \leq i < n$ (for some $n \geq 2$) the definition of every A_i cites an instance of A_{i+1} , and the definition of A_n cites an instance of A_1 .

In this chapter we are interested in notions for which a recursive definition is elegant and convenient. Important examples will indeed include recursive routines, recursive syntax definitions (of which we have already seen examples), and recursive data structures.

One class of recursive data structures, the *tree* in its various guises, appears in many applications and embodies the very idea of recursion. It's also studied in this chapter.

16.1 BASIC EXAMPLES

At this stage you may be wondering whether a recursive definition makes any sense at all. How can we define a concept in terms of itself? Does such a definition mean anything at all, or is it just a vicious circle?

It's right to wonder. Not all recursive definitions define anything at all. When you ask for a description of someone and all you get is "*Sarah? She is just Sarah, what else can I say?*" you're not learning much. So we'll have to look for criteria that guarantee that a definition is useful even if recursive.

Before we do this, however, let's convince ourselves in a more pragmatic way by looking at typical examples where recursion is obviously useful and seems, just as obviously, to make sense. This will give us a firm belief — little more than a belief indeed, the kind that's based on hope and a prayer — that recursion is a practically useful way to define grammars, data structures and algorithms. Then it will be time to look for a proper mathematical basis on which to establish recursive definitions.

→ "[MAKING SENSE OF RECURSION](#)", 16.5, page 384.

Recursive definitions

With the introduction of genericity, we were able to define a *type* as either:

- T1 • A non-generic class, such as *INTEGER* or *METRO_STATION*.
- T2 • A generic derivation, of the form $C [T]$, where C is a generic class and T is a *type*.

← "[Definition: Class type](#)", page 247.

This is a recursive definition; it simply means, using the generic classes *ARRAY* and *LIST*, that valid classes are:

- *INTEGER*, *METRO_STATION* and such: non-generic classes, per case **T1**.
- Through case **T2**, direct generic derivations: *ARRAY [INTEGER]*, *LIST [METRO_STATION]* etc.
- Applying **T2** again, recursively: *ARRAY [LIST [INTEGER]]*, *ARRAY [ARRAY [LIST [METRO_STATION]]]* and so on: generic derivations at any level of nesting.

You may consider using a similar technique to answer the exercise which, in the first chapter, asked you to define "alphabetical order".

← [1-E.3, page 16](#).

Recursively defined grammars

Consider an Eiffel subset with just two kinds of instruction:

- Assignment, of the usual form *variable* := *expression*, but treated here as a terminal since we don't need to specify it further.
- Conditional, with only a **then** part (no **else**) for simplicity.

A grammar defining this language is:

```

Instruction  $\triangleq$  Assignment | Conditional
Conditional  $\triangleq$  if Condition then Instruction end
  
```

For our immediate purposes **Condition** is, like **Assignment**, a terminal. This grammar is recursive, since the definition of **Instruction** involves **Conditional** as one of the choices, and **Conditional**, in turn involves **Instruction** as part of the aggregate. But since there's a non-recursive alternative, **Assignment**, the grammar productions clearly specify what an instruction may look like:

- Just an assignment.
- A **Conditional** containing an assignment: **if c then a end**.
- The same with any degree of nesting: **if c₁ then if c₂ then a end end**, **if c₁ then if c₂ then if c₃ then a end end end** and so on.

Recursive grammars are indeed an indispensable tool for any language that — like all significant programming languages — supports nested structures.

Recursively defined data structures

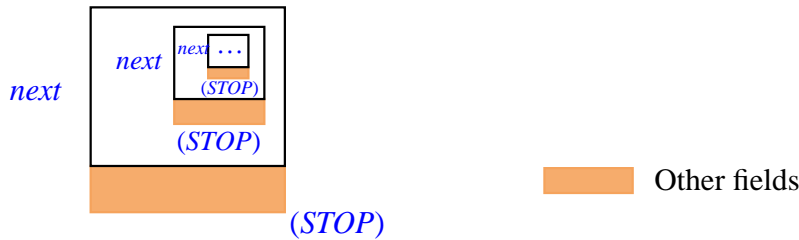
The class *METRO_STOP* represented the notion of stop in a metro line:

← Page 124.

```

class STOP create
  ...
feature
  next: STOP
    -- Next stop on same line
  ... Other features omitted (see page 124) ...
end
  
```

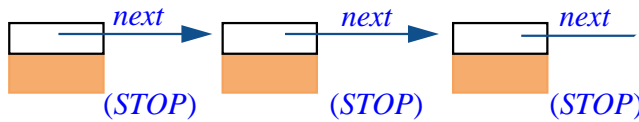
A naïve interpretation would deduce that every instance of *STOP* contains an instance of *STOP*, which itself contains another ad infinitum, as in a Laughing Cow scheme. This would indeed be the case if *STOP* were an expanded type:



*Nested fields
(not the correct
interpretation)*

This is impossible, however, and *STOP* is, in any case, a **reference** type, like any other type defined as **class X ...** with no other qualification. So the real picture is the one originally shown:

← Page 118.



A linked line

Recursion in such a data structure definition simply indicates that every instance of the class contains a reference to a potential instance of the same class — “potential” because the reference may be void, as for the last stop in the figure.

In the same chapter we encountered another example of self-referential class definition: a class *PERSON* with an attribute *spouse* of type *PERSON*.

This is a very common case in definitions of useful data structures. From linked lists to *trees* of various kinds (such as the binary trees studied later in this chapter), the definition of many useful object types includes references to objects of the type being defined, or (indirect recursion) a type that depends on it.

Recursively defined algorithms

The famous Fibonacci sequence, enjoying many beautiful properties and many applications to mathematics and the natural sciences, has the following definition:

$$\begin{aligned}
 F_0 &= 0 \\
 F_1 &= 1 \\
 F_i &= F_{i-1} + F_{i-2} \quad \text{-- For } i > 1
 \end{aligned}$$

Touch of History: Fibonacci's rabbits

Leonardo Fibonacci from Pisa (1170-1250) played the key role in making Indian and Arab mathematics known to the West and, through many contributions of his own, helping to start modern mathematics. He stated like this the problem that leads to his famous sequence (which was already known to Indian mathematicians):

About Fibonacci:
www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html; about the sequence: www.gap.dcs.st-and.ac.uk/~history/Mathematicians/Fibonacci.html

A man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if every month each pair begets a new pair which from the second month on becomes productive?

The answer is that the pairs at month i include those already present at month $i - 1$ (no rabbits die), numbering F_{i-1} , plus those begot by pairs already present at month $i - 2$ (since pairs are fertile starting the second month), numbering F_{i-2} . This gives the above formula; successive values are 0, 1, 1, 2, 3, 5, 8 and so on, each the sum of the previous two.

The formula readily yields a recursive routine computing F_n for any n :

```
fibonacci (n: INTEGER): INTEGER is
    -- Element of index n in the Fibonacci sequence
    require
        non_negative: n >= 0
    do
        if n = 0 then
            Result := 0
        elseif n = 1 then
            Result := 1
        else
            Result := fibonacci (n - 1) + fibonacci (n - 2)
        end
    end
end
```



Fibonacci

Programming Time! **Recursive Fibonacci**

Write a small system that includes the above recursive routine and prints out its result. Try it for a few values of n — including 12, as in Fibonacci's original riddle — and verify that the results match the expected values.

The function includes two recursive calls, highlighted. That it works at all may look a bit mysterious (that's why it's good to check it for a few values); as you progress through this chapter the legitimacy of such recursively defined routines should become increasingly convincing.

The prime argument in favor of writing the function this way is that it elegantly matches the original, mathematical definition of the Fibonacci sequence. On further look, however, it's not that exciting, because a non-recursive version is also easy to obtain.

Programming Time! **Non-recursive Fibonacci**

Can you write (without first looking at the solution overleaf) a function that computes any Fibonacci number, using a loop rather than recursion?

The following function indeed yields the same result as the above *fibonacci* (try it too for a few values):

```

fibonacci1 (n: INTEGER): INTEGER is
  -- Element of index n in the Fibonacci sequence
  -- (non-recursive version)
  require
    positive: n >= 1
  local
    i, previous, second_previous: INTEGER
  do
    from
      i := 1 ; Result := 1
    invariant
      Result = fibonacci (i)
      previous = fibonacci (i - 1)
    variant
      n - i
    until i = n loop
      i := i + 1
      second_previous := previous
      previous := Result
      Result := previous + second_previous
    end
  end
end

```

For convenience this version assumes $n \geq 1$ rather than $n \geq 0$. Thanks to the initialization rules *previous* starts out as zero, ensuring the initial satisfaction of the invariant since $F_0 = 0$. The variable *second_previous* is set anew in each loop iteration and doesn't need specific initialization.

This version, just a trifle more remote from the original mathematical definition, is still simple and clear (note in particular the loop invariant). Some may still prefer the recursive version, but it's largely a matter of taste especially since (as we'll see) that version might, depending on the compiler, be less efficient at run time.

Taste and efficiency aside, if it were only for such examples we would have a hard time convincing ourselves of the indispensability of recursive routines. We need cases in which recursion provides a definite plus, for example because any non-recursive competitor is significantly more complex.

There are indeed many such problems. One that concentrates many of the interesting properties of recursion with the least irrelevant detail arises from an attractive puzzle: the Tower of Hanoi.

16.2 THE TOWER OF HANOI

In the great temple of Benares, under the dome that marks the center of the world, three needles of diamond are set on top of a brass plate. Each needle is a cubit high, and thick as the body of a bee. One one of these needles God strung, at the beginning of ages, sixty-four disks of pure gold; the largest disk rests on the brass and the others, ever smaller, rest over each other all the way to the top. It is the sacred tower of Brahma.

Night and day the priests, following one another on the steps of the altar, work to transfer the tower from the first diamond needle to the third, without deviating from the rules just stated, set by Brahma. When all is over, the tower and the Brahmins will fall, and it will be the end of the worlds.



*Tower of Hanoi
(or should it be
Benares?) with
9 disks, initial
state*

In spite of its oriental veneer, this story is the creation of the French mathematician Édouard Lucas (signing as “N. Claus de Siam”, anagram of “Lucas d’Amiens”, after his native city). On a market in Thailand — Siam indeed — I bought the above rendition of his tower. The labels **A**, **B** and **C** are my addition. I won’t expand on why I choose a model made of wood rather than diamond, gold and brass, but it is legitimate, since I did have a large suitcase, to ask why it has only nine disks:

Quiz time! **Hanoi tower size**

Why do commercially available models of the Towers of Hanoi puzzle have far fewer than 64 disks?

(Hint: the game comes with a small sheet of paper listing a solution to the puzzle, in the form of a sequence of moves: **A** to **C**, **A** to **B** etc.)

To answer this question, let's try to assess the minimal number H_n of individual “move” operations required — if there is a solution — to transfer n disks from needle **A** to needle **B**, using needle **C** as intermediate storage and following the rules of the game; n is 64 in the original version and 9 for the small model.

We observe that any strategy for moving n disks from **A** to **B** must at some point move the largest disk from **A** to **B**. This is only possible, however, if needle **B** is free of *any* disks at all, and **A** contains only the largest disk, all others having been moved to **C** — since there is no other place for them to go:



Intermediate state

What is the minimum number of moves to reach this intermediate situation? We must have transferred $n - 1$ disks (all but the largest) from **A** to **C**, using **B** as intermediate storage; the largest disk, which must stay on **A**, plays no role in this operation. The problem is symmetric between **B** and **C**; so the minimum number of moves to achieve the intermediate situation is H_{n-1} .

Once we have reached that situation, we must move the largest disk from **A** to **B**; it remains then to transfer the $n - 1$ smaller disks from **C** to **B**. In all, the minimum number of moves H_n for transferring n disks, for $n > 0$, is

$$H_n = 2 * H_{n-1} + 1$$

(H_{n-1} moves to transfer $n - 1$ disks from **A** to **C**, one move to take the largest disk from **A** to **B**, and H_{n-1} again to transfer the $n - 1$ smaller disks from **A** to **C**). Since $H_0 = 0$, this gives

$$H_n = 2^n - 1$$

and, as a consequence, the answer to our quiz: remembering that 2^{10} (that is, 1024) is over 10^3 , we note that 2^{64} is over $1.5 \cdot 10^{19}$; that's a lot of moves.

A year is around 30 million seconds. At one second per move — very efficient priests — the world will collapse in about 500 billion years, close to 50 times the estimated age of the universe.

This reasoning for the evaluation of H_n was *constructive*, in the sense that it also gives us a **practical strategy** for moving n disks (for $n > 0$) from **A** to **B** using **C** as intermediate storage:

- Move $n - 1$ disks from **A** to **C**, using **B** as intermediate storage, and respecting the rules of the game.
- Then **B** will be empty of any disk, and **A** will only have the largest disk; transfer that disk from **A** to **B**. This respects the rules of the game since we are moving a single disk, from the top of a needle, to an empty needle.
- Then move $n - 1$ disks from **C** to **B**, using **A** as intermediate storage, respecting the rules of the game; **B** has one disk, but it will not cause any violation of the rules of the game, since it is larger than all the ones we want to transfer.

This strategy turns the number of moves $H_n = 2^n - 1$ from a theoretical minimum into a practically achievable goal. We may express it as a recursive routine, part of a class *NEEDLES*:

```

hanoi (n: INTEGER; source, target, other: CHARACTER) is
  -- Transfer n disks from source to target, using other as
  -- intermediate storage, according to the rules of the
  -- Tower of Hanoi puzzle.
  require
    non_negative: n >= 0
    different1: source /= target
    different2: target /= other
    different3: source /= other
  do
    if n > 0 then
      hanoi (n-1, source, other, target)
      move (source, target)
      hanoi (n-1, other, target, source)
    end
  end
end

```

The discussion of contracts for recursive routines will add more precondition clauses and a postcondition.

→ “*CONTRACTS FOR RECURSIVE ROUTINES*”, 16.6, page 395.

By convention, the needles are represented by characters, as in 'A', 'B', 'C'. The routine contains two recursive calls, highlighted.

The basic operation *move* (*source*, *target*) moves a single disk, the top one on needle *source*, to needle *target*; its precondition is that there is at least one disk on *source*, and that on *target* either there is no disk or the top disk is larger than the top disk on *source*. If you have access to the wireless network of the Great Temple of Benares you can program *move* to send an SMS to the cell phone or Bluetooth-enabled PDA of the appropriate priest, directing him to move a disk from *source* to *target*. For the rest of us you can write *move* as a procedure that displays a one-disk-move instruction in the console:

```

move (source, target: CHARACTER) is
  -- Prescribe move from source to target.
  do
    io.put_character (source)
    io.put_string (" to ")
    io.put_character (target)
    io.put_new_line
  end

```

Programming Time! **The Tower of Hanoi**

Write a system with a root class *NEEDLES* including the procedures *hanoi* and *move* as shown. Try it for a few values of *n*.

For example executing the call

```
hanoi (4, 'A', 'B', 'C')
```

will print out the sequence of fifteen ($2^4 - 1$) instructions

A to C	B to C	B to A
A to B	A to C	C to B
C to B	A to B	A to C
A to C	C to B	A to B
B to A	C to A	C to B

Shown here split into three columns; read it column by column, top to bottom. The move of the biggest disk has been highlighted.

which indeed moves four disks successfully from **A** to **B**, respecting the rules of the game.

One way to look at the recursive solution — procedure *hanoi* — is that it works as if it were permitted to move the top $n-1$ disks all at once to a needle that has either no disk, or only the biggest disk. In that case we would start by performing this operation from *source* to *other* (here **A** to **C**):



Fictitious initial global move

Then we would move the biggest disk from **A** to **B**, our final *target*; this single-disk move is clearly legal since there's nothing on **B**. Finally we would again perform a global move of $n-1$ disks from **C**, where we've parked them, to **B**, which is OK because they are in order and the largest of them is smaller than the disk now on **B**.

Of course this is a fiction since we are only permitted to move one disk at a time, but to move $n-1$ disks we may simply apply the same technique recursively, knowing that the target needle is either empty or occupied by a disk larger than all those we manipulate in this recursive application. If $n = 0$, we have nothing to do.

Don't be misled by the apparent frivolity of the Tower of Hanoi example. The solution serves as a model for many recursive programs with important practical applications. The simplicity of the algorithm, resulting from the use of two recursive calls, makes it an ideal workbench to study the properties of recursive algorithms, as we'll do when we return to it later in this chapter.

16.3 BINARY TREES

If the Tower of Hanoi solution is the quintessential example of a recursive routine, the binary tree is the quintessential example of a recursive data structure. We may define it as follows:

Definition: binary tree

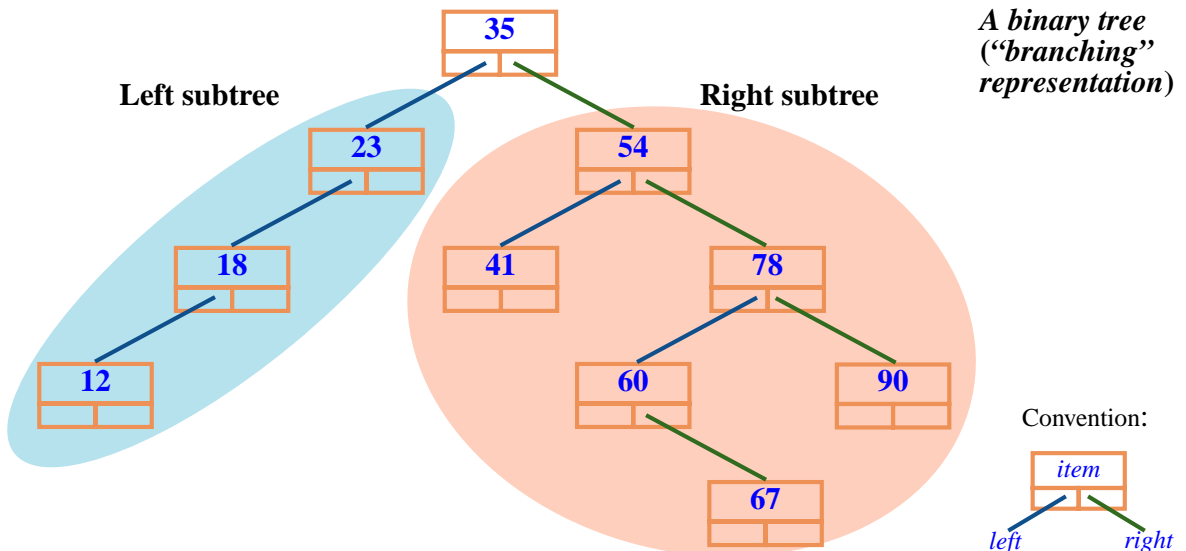
A binary tree over G , for an arbitrary data type G , is a finite set of items called **nodes**, each containing a value of type G , such that the nodes, if any, are divided into three disjoint parts:

- A single node, called the **root** of the binary tree.
- (Recursively) two **binary trees** over G , called the *left subtree* and *right subtree*.

It's easy to express this as a class skeleton, with no routines yet:

```
class BINARY_TREE [G] feature
  item: G
  left, right: BINARY_TREE [G]
end
```

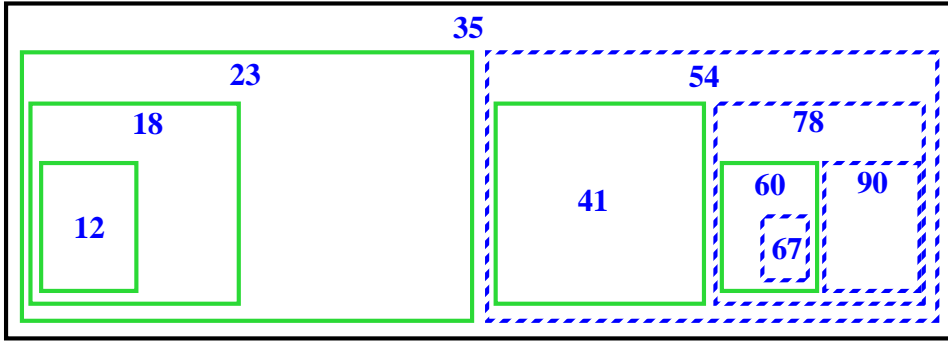
where a void reference indicates an empty binary tree. We may illustrate a binary tree — here over *INTEGER* — as follows:



A binary tree
("branching"
representation)

This “branching” form is the most common style of representing a binary tree, but not the only one; as in the case of abstract syntax trees, we might also opt for a *nested* representation, which for this example would look like this:

← “*NESTING AND THE SYNTAX STRUCTURE*”, page 44.



A binary tree in nested representation

Convention:
 Left subtree
 Right subtree

The definition explicitly allows a binary tree to be empty (“the nodes, if any”). Without this, of course, the recursive definition would leave to an infinite structure, whereas our binary trees are, as the definition also prescribes, finite.

If not empty, a binary tree may have: no subtree; a left subtree only; a right subtree only; or both. This property also holds anywhere in the tree, but to express it more generally we need the notion of “child” and “parent”.

A recursive routine on a recursive data structure

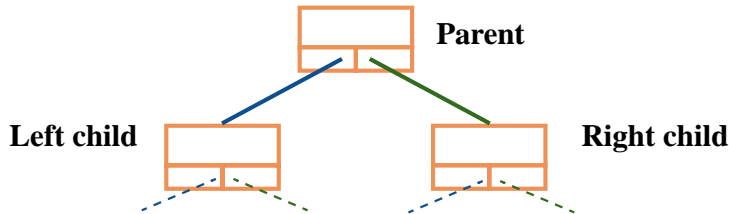
Many routines of a class that defines a data structure recursively will follow the definition’s recursive structure. A simple example is a routine computing the number of nodes in a binary tree. The node count of an empty tree is zero; the node count of a non-empty tree is one — corresponding to the root — plus (recursively) the **node counts** of the left and right subtrees, if any. We may use this observation to include a recursive function in the class *BINARY_TREE*:

```
count: INTEGER is
  -- Number of nodes
do
  Result := 1
  if left /= Void then Result := Result + left.count end
  if right /= Void then Result := Result + right.count end
end
```

Note the similarity of the recursive structure to procedure *Hanoi*.

Children and parents

The **children** of a node — nodes themselves — are the root nodes of its left and right subtrees, if any:



*A binary tree
("branching"
representation)*

If C is a child of B , then B is the **parent** of C . "The" parent because of the following result:

Theorem: Single Parent

Every node in a binary tree has exactly one parent, except for the root which has no parent.

The theorem seems obvious from the picture, but we have to prove it; this gives us an opportunity to encounter *recursive proofs*.

Recursive proofs

The recursive proof of the Single Parent theorem follows the structure of the recursive definition.

Unless it is empty (in which case the theorem trivially holds) a binary tree consists of a root and two disjoint binary trees; we assume — this is the "recursion hypothesis" — that they satisfy the theorem. From the definitions of "binary tree", "child" and "parent" it follows that a node C may have a parent P in the binary tree only through one of the following three ways:

- P1 • P is the root of the binary tree, and C is the root of either its left subtree or its right subtree.
- P2 • They both belong to the left subtree, and P is the parent of C in that subtree.
- P3 • They both belong to the right subtree, and P is the parent of C in that subtree.

In case **P1**, C has, from the recursion hypothesis, no parent in its subtree; so it has one parent, the root, in the binary tree as a whole. In cases **P2** and **P3**, again by the recursion hypothesis, P was the single parent of C in their respective subtree, and this is still the case in the whole tree.

Any node C other than the root fall into one of these three cases, and hence has exactly one parent. In none of these cases can C be the root which, as a consequence, has no parent. This completes the proof.

Recursive proofs of this kind are useful when you need to establish that a certain property holds for all instances of a recursively defined concept. The structure of the proof follows the structure of the definition:

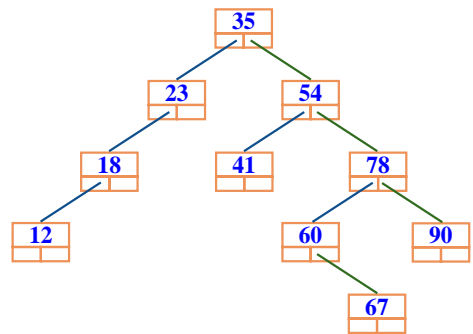
- For any non-recursive case of the definition, you must prove the property directly. (In the example the non-recursive case is an empty tree.)
- A case of the definition is recursive if it defines a new instance of the concept in terms of existing instances. For those cases you may assume that the property holds of these instances (this is the *recursion hypothesis*) to prove that it holds of the new one.

This technique applies to recursively defined concepts in general. We'll see its application to recursively defined routines such as *hanoi*.

More binary tree properties and terminology

A node of a binary tree may have:

- Both a left child and a right child, like the top node, labeled **35**, of our example.
- Only a left child, like all the nodes of the left subtree, labeled **23**, **18**, **12**.
- Only a right child, like the node labeled **60**.
- No child, in which case it is called a **leaf**. In the example the leaves are labeled **12**, **41**, **67** and **90**.



(From the figure on page 370.)

We define an **upward path** in a binary tree as a sequence of zero or more nodes, where any node in the sequence is the parent of the previous one if any. In our example, the nodes of labels **60**, **78**, **54** form an upward path. We have the following property, a consequence of the Single Parent theorem:

Theorem: Root Path

From any node of a binary tree, there is a single upward path to the root.

Proof: consider an arbitrary node C and the upward path starting at C and obtaining by adding the parent of each node on the path, as long as there is one; the Single Parent theorem ensures that this path is uniquely defined. If the path

is finite, its last element is the root, since any other node has a parent and hence would allow us to add one more element to the path; so to prove the theorem it suffices to show that all paths are finite.

The only way for a path to be infinite, since our binary trees are finite sets of nodes, would be to include a **cycle**, that is to say if a node n appeared twice (and hence an infinite number of times). This means the path includes a subsequence of the form $n \dots n$. But then n appears in its own left or right subtree, which is impossible from the definition of binary trees.

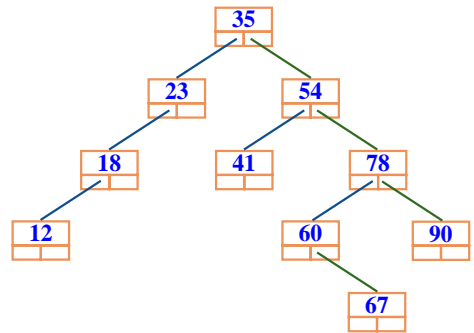
Considering downward rather than upward paths gives an immediate consequence of the preceding theorem::

Theorem: Downward Path

For any node of a binary tree, there is a single downward path connecting the root to the node through successive applications of *left* and *right* links.

The **height** of a binary tree is the maximum number of nodes on a downward path from the root to a leaf. In our basic example, reproduced again on the side, the height is 5, obtained through the path from the root to the leaf labeled **67**.

It is possible to define this notion recursively, following again the recursive structure of the definition of binary trees: the height of an empty tree is zero; the height of a non-empty tree is one plus the maximum of (recursively) the heights of its two subtrees. We may add the corresponding function to class *BINARY_TREE*:



```

height: INTEGER is
  -- Maximum number of nodes on a downward path
  local
    lh, rh: INTEGER
  do
    if left /= Void then lh := left.height end
    if right /= Void then rh := right.height end
    Result := 1 + lh.max(rh)
  end
  
```

$x \bullet \max(y)$ is the maximum of x and y .

This adapts the recursive definition to the convention used by the class, which only considers non-empty binary trees, although either or both subtrees, *left* and *right*, may be empty. Note again the similarity to *hanoi*.

Binary tree operations

Class *BINARY_TREE* as given so far only has queries. The following two procedures, to be added to the class, let us add a left or right child:

```

add_left (x: G) is
    -- Create left child of value x
    require
        no_left_child_behind: left = Void
    do
        create left.make (x)
    end

add_right ... Same model...

```

using the creation procedure

```

make (x: G) is
    -- Initialize with item value x.
    do
        item := x
    ensure
        set: item = x
    end

```

Traversals

Being defined recursively, binary trees, not surprisingly, lead to many recursive routines. Function *height* was one; here is another. Assume that you are requested to print all the *item* values associated with nodes of the tree. The following procedure, to be added to the class, does the job:

```

print_all is
    -- Print all node values
    do
        if left /= Void then print_all (left) end
        print (item)
        if right /= Void then print_all (right) end
    end

```

This uses the procedure *print* (available to all classes through their common ancestor *ANY*) which prints a suitable representation of a value of any type; here the type is *G*, the generic parameter in *BINARY_TREE [G]*.

The structure is in line with *hanoi* and the preceding binary tree routines.

Although the business of *print_all* is to print every node item, the algorithm scheme is independent of the specific operation, here *print*, that we perform on *item*. The procedure is an example of a binary tree **traversal**: an algorithm that performs a certain operation once on every element of a data structure, in a precisely specified order.

For binary trees, three such traversal orders are often useful:

Binary tree traversal orders

- **Inorder**: traverse left subtree, visit root, traverse right subtree.
- **Preorder**: visit root, traverse left, traverse right.
- **Postorder**: traverse left, traverse right, visit root.

In these definitions, “*visit*” means performing the individual node operation, such as *print* in the *print_all* example; “*traverse*” means a recursive application of the algorithm to a subtree, or no action if the subtree is empty.

The procedure *print_all* is an illustration of inorder traversal. We may easily express the other two variants in the same recursive form; for example, a routine *post* for postorder traversal will have the routine body

```

if left /= Void then post (left) end
if right /= Void then post (right) end
visit (item)

```

where *visit* is the node operation, such as *print*.

In the quest for software reuse, it is undesirable to write a different routine for variants of a given traversal scheme just because the *visit* operation changes. To avoid this, we may use the operation itself as an argument to the traversal routine. This will be possible through the notion of **agent** in a later chapter.

→ Chapter 20,
Event-driven design.

Binary search trees

For a general binary tree, procedure *print_all*, implementing inorder traversal, prints the node values in an arbitrary order. For the order to be significant, we must move on from binary trees to binary *search* trees.

The set G over which a general binary tree is defined can be any set. For binary search trees, we assume that G is equipped with a **total order relation enabling us** to compare two arbitrary elements of G with the boolean expression $a < b$, such that exactly one of $a < b$, $b < a$ and *equal* (a, b) is true. Examples of such sets include *INTEGER* or *REAL*, with the usual $<$ relation, but G could be any other set on which we know a total order.

→ We'll learn more on total orders in the study of topological sort: "[Total orders](#)", page 418.

As usual we write $a \leq b$ for ($a < b$) **or** *equal* (a, b), and $a > b$ for $b < a$.

Over such totally ordered sets we may define binary search trees:

Definition: binary search tree

A binary search tree over a totally ordered set G is a binary tree over G such that, for any subtree of root *item* value r :

- The item value le of any node in the left subtree satisfies $le < r$.
- The item value ri of any node in the right subtree satisfies $ri > r$.

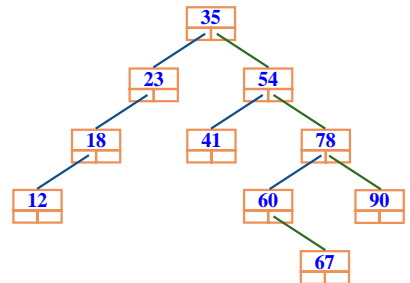
The node values in the left subtree are less than the value for the root, and those in the right subtree are greater; this property must apply not only to the tree as a whole but also, recursively, to any of its immediate or indirect subtrees. We will call it the **Binary Search Tree Invariant**.

This definition implies that all the *item* values of the tree's node are different. This is the convention that we will take, largely for simplicity. It is also possible to accept duplications; then the conditions in the definitions become $le \leq r$ and $r \leq ri$. An *exercise* asks you accordingly to adapt the binary search tree algorithms that we are going to see.

→ Exercise [16-E.3](#), page 405.

Our example binary tree of integers is a binary search tree: all the values in the left subtree are less than the root value, **35**, all those in the right subtree are greater, and again recursively in every subtree.

The procedure *print_all*, applied to a binary search tree, will print all the node items in order, from smallest to greatest.



Programming Time! Printing values in order

Using the procedures given so far, write a program that builds the example tree, then prints the node items using `print_all`. Check that the values are in order.

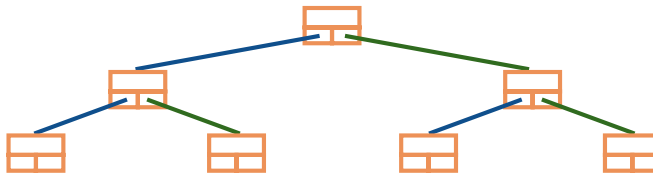
Performance

Let's look more closely at why binary search trees are useful as container structures — a potential competitor to hash tables. The reason is that they usually provide much better performance than sequential lists. Assuming random data, a sequential list provides us with

- $O(1)$ insertion (if we keep the elements in the order of insertion)
- $O(n)$ search

With a binary search tree, both operations can be $O(\log n)$, which is much better than $O(n)$ for large n . (Remember that in big- O notation it doesn't matter what base we choose for the logarithms.) Here is the analysis for a **full** binary tree, that is to say one in which both subtrees of any given node have exactly the same height h :

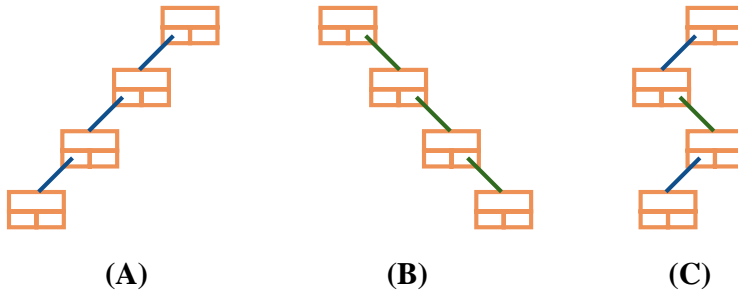
← The performance tables were on pages [281](#) and [282](#).



A full binary tree

It is clear, by induction on h , that the number of nodes n in a full tree of height h is $2^h - 1$ (in the above figure, h is 3 and n is 7). This implies that for a given number of nodes n the height is $\log_2(n + 1)$, which is $O(\log n)$. In a full tree, both a search and an insertion — using algorithms given below, which you can already guess — will start from the root and go to a leaf, taking $O(\log n)$ time. This is the major attraction of binary search trees.

Of course most practical binary trees are not full; if you are out of luck with the order of insertion, the performance can in fact be as bad as with sequential lists, $O(n)$ — with added storage costs since each node has both a *left* field and a *right* field where a linked list cell has just one. The following figure shows such cases: insertions in descending order (A), ascending order (B), greatest then smallest then second greatest and so on (C).



Some binary search tree schemes causing $O(n)$ behavior

With a random enough order of insertions, however, the binary search tree will remain sufficiently close to full to ensure $O(\log n)$ behavior. You can actually *guarantee* $O(\log n)$ insertions, searches and deletions by using the **AVL** variant of binary search trees, which remain near-full (or “balanced”), as detailed in the [appendix](#) to this chapter.

→ “[APPENDIX: AVL TREES](#)”, 16.10, page 403.

Inserting, searching, deleting

Here is a recursive routine for searching a binary search tree (again to be added to class `BINARY_TREE`, as the following ones):

```

has (x: G): BOOLEAN is
    -- Does x appear in any node?
    require
        argument_exists: x /= Void
    do
        if x = item then
            Result := True
        elseif x < item and left /= Void then
            Result := left.has (x)
        elseif x > item and right /= Void then
            Result := right.has (x)
        end
    end
end

```

There is no **else** clause: if none of the conditions hold the result will be false. The algorithm is $O(h)$ where h is the height of the tree, meaning $O(\log n)$ for full or near-full trees.

A non-recursive version, using a loop, is also possible:

```

has1 (x: G): BOOLEAN is
    -- Does x appear in any node?
require
    argument_exists: x /= Void
local
    node: BINARY_TREE [G]
do
    from
        node := Current
    until
        Result or node = Void
    invariant
        -- x doesn't appear above node on downward path from root
    variant
        -- (Height of tree) – (Length of path from root to node)

    loop
        if x < item then
            node := left
        elseif x > item then
            node := right
        else
            Result := True
        end
    end
end

```

For *inserting* an element, we may use the following recursive procedure::

```

put (x: G) is
    -- Insert x if not already present.
require
    argument_exists: x /= Void
do
    if x < item then
        if left = Void then
            add_left (x)
        else
            left.put (x)
        end
    end

```

← About *add_left* and *add_right* see page 375.

```

elseif  $x > item$  then
    if  $right = Void$  then
        add_right ( $x$ )
    else
        right.put ( $x$ )
    end
end
end
end

```

The absence of an **else** clause for the outermost **if** reflects the decision to ban duplicate information. The non-recursive version is left as an exercise.

← See page 377.

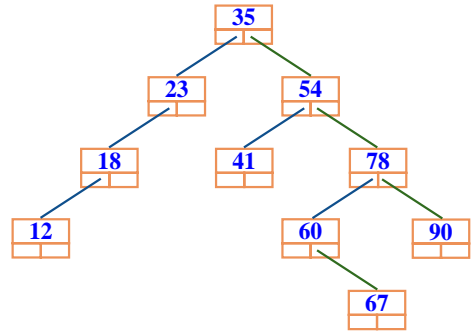
→ 16-E.4, page 405.

The next natural step after insertion is to write a deletion procedure *remove* ($x: G$). This is less simple because we can't just remove the node containing x (unless it's a leaf and not the root, in which case we make the corresponding *left* or *right* reference void); we can't either leave an arbitrary value there since it would destroy the Binary Search Tree Invariant.

Actually we could put a special boolean attribute in every node, indicating whether the *item* value is meaningful, but that makes things too complicated and affects the other algorithms.

What we should do is reorganize the node values, moving up some of those found in subtrees of the node where we find x to reestablish the Binary Search Tree Invariant. In the example binary search tree, a call *remove* (35), affecting the value in the root node, might either:

- Move up all the values in the left subtree (where each node has a single child, on the left).
- Move up the value in the right child, 54, then recursively apply a similar choice to move up values in one of its subtrees.



(From the figure on page 370.)

Like search and insertion, the process should be $O(h)$ where h is the height of the tree. Since this is particularly interesting in the case of AVL (balanced) trees the algorithm will appear in the corresponding section, but since you have all the elements to write a first version I strongly suggest you try your hand at it now, following the inspiration of the preceding routines:

Programming Time! Deletion in a binary search tree

Write a procedure *remove* ($x: G$) that removes from a binary search tree the node, if any, of item value x , preserving the Binary Search tree Invariant.

16.4 FROM LOOPS TO RECURSION

We have seen that some recursive algorithms — Fibonacci numbers, search and insertion for binary search trees — have a loop equivalent. What about the other way around?

It is indeed not hard to replace *any* loop by a recursive routine. Consider an arbitrary loop, given here without its invariant and variant (although we will see their recursive counterparts later):

```
from Init until Exit loop Body end
```

We may replace it by

```
Init  
loop_equiv
```

with the procedure

```
loop_equiv is  
  --Emulate a loop of exit condition Exit and body Body.  
  do  
    if not Exit then  
      Body  
      Loop_equiv  
    end  
  end
```

In some programming languages known as **functional languages** (the best known are Lisp, Scheme, Haskell, ML) this recursive form is the preferred style, even if loops are available for convenience. We could use it too in our framework, replacing for example the first complete example of the discussion of loops, which animated a Metro line by moving a red dot, by

← Page 166.

```
Line8.start  
animate_rest (Line8)
```


with the auxiliary routine

```

animate_rest (line: METRO_LINE) is
  -- Animate stations of line from current cursor position on
  do
    if not line.after then
      show_spot (line.item.location)
      line.forth
      animate_rest (line)
    end
  end
end

```

(A more complete version would restore the cursor to its original position.)

The recursive version is elegant, but there is no particular reason in our framework to prefer it to the loop form; indeed we will continue to use loops.

The conclusion might be different if we were using functional programming languages, where systematic reliance on recursive routines is part of a distinctive style of programming. You should definitely, as part of your further programming education, take a course in functional programming, and discover for yourself the inner consistency of that approach.

Even if just for theoretical purposes, it's interesting to know that loops are conceptually not needed if we have routines that can be recursive. As an example, recursion gives us a more concise version of the loop-based routine *paradox* demonstrating the unsolvability of the Halting Problem:

← ["AN APPLICATION: PROVING THE UNDECIDABILITY OF THE HALTING PROBLEM", page](#)

```

recursive_paradox is
  -- Terminate if and only if not.
  do
    if terminates ("C:\your_project") then
      recursive_paradox
    end
  end
end

```

Knowing that we can easily emulate loops with recursion, it's natural to ask about the reverse transformation. Do we really need recursive routines, or could we use loops instead? We've seen the straightforward cases mentioned above (*has* and *put* for binary search trees, *Fibonacci*) but others such *hanoi*, *height*, *print_all* don't seem to have an obvious recursion-free equivalent. To understand what exactly can be done we have to look more closely into the meaning and properties of recursive routines.

16.5 MAKING SENSE OF RECURSION

The experience of our first few recursive schemes allows us to probe a bit deeper into the meaning of recursive definitions.

Vicious circle?

First let's go back to the impolite but basic question: does a recursive definition mean anything at all? The examples, especially those of recursive routines, should by now — I hope — be sufficiently convincing to suggest a positive answer, but we should still retain a healthy dose of doubt. After all we are never very far from definitions that make no sense at all — vicious circles. With recursion we try to define a concept in terms of itself, but we can't just define it *as* itself. If I say

“Computer science is the study of computer science”

I have not defined anything at all, just stated a tautology; not one of those tautologies of logic, which are things to prove and hence possibly interesting, just a platitude. If I refine this into

← “*Definition: Tautology*”, page 80.

“Computer science is the study of programming, data structures, algorithms, applications, theories and other areas of computer science”

I have added some usable elements but still not produced a satisfactory definition. Recursive routines can, similarly, be obviously useless, as:

```
p (x: INTEGER) is
    -- What good is this?
    do p (x) end
```

which for any value of the argument would execute forever, never producing any result.

We'll see that “forever” in this case means, for a typical compiler's implementation of recursion on an actual computer, “until the stack overflows and causes the program to crash”. So in practice, given the speed of computers, “forever” doesn't last very long.

How do we avoid such obvious misuses of recursion? If we try to understand why the recursive definitions seen so far seem intuitively to make sense, we can nail down three interesting properties:

Touch of Methodology: Well-formed recursive definition

A useful recursive definition should ensure that:

- R1 • There is at least one non-recursive branch.
- R2 • Every recursive branch occurs in a context that differs from the original.
- R3 • For every recursive branch, the change of context (R2) brings it closer to at least one of the non-recursive cases (R1).

For a recursive routine, the change of “context” (R2, R3) may be that the call uses a different argument, as will a call $r(n-1)$ in a routine $r(n: \text{INTEGER})$; that it applies to a different target, as in a call $x.r(n)$ where x is not the current object; or that it occurs after the routine has changed at least one object field.

The recursive routines seen so far satisfy these requirements:

- the body of Hanoi (n, \dots) is of the form **if** $n > 0$ **then** ... **end** where the recursive calls are in the **then** part, but there is no **else** part, so the routine does nothing for $n = 0$ (R1). The recursive calls are of the form Hanoi ($n-1, \dots$), changing the first argument. and also switching the order of the others (R2). Replacing n by $n-1$ brings the context closer to the non-recursive case $n = 0$ (R3). ← Page [367](#).
- The recursive has for binary search trees has non-recursive cases for $x = \text{item}$, as well as for $x < \text{item}$ if there is no left subtree, and $x > \text{item}$ if there is no right subtree (R1). It calls itself recursively on a different target, left or right rather than the current object (R2); every such call goes to the left or right subtree, closer to the leaves where the recursion terminates (R3). The same scheme governs other recursive routines on binary trees, such as height. ← Page [379](#).
- The recursive version of the metro line traversal, animate rest, has a non-recursive branch (R1), doing nothing, for a cursor that’s after. The recursive call doesn’t change the argument, but it is preceded by a call line.forth which changes the state of the line list (R2), moving the cursor closer to a state satisfying after and hence to the non-recursive case (R3). ← Page [374](#).
- The recursive version of the metro line traversal, animate rest, has a non-recursive branch (R1), doing nothing, for a cursor that’s after. The recursive call doesn’t change the argument, but it is preceded by a call line.forth which changes the state of the line list (R2), moving the cursor closer to a state satisfying after and hence to the non-recursive case (R3). ← Page [383](#).

R1 and R2 also hold for recursive definitions of concepts other than routines:

- The mini-grammar for Instruction has the non-recursive case Assignment. ← Page [361](#).
- All our recursively defined data structures, such as STOP, are recursive through references (never expanded values), and references can be void. Void values indeed serve as terminators of linked structures. ← Page [361](#).

In the case of recursive routines, combining the above three rules suggests a notion of **variant** similar to the loop variants through which we guarantee that loops terminate:

← “Loop termination”, page 159.

Touch of Methodology: Recursion Variant

Every recursive routine should be declared with an associated recursion variant, an integer quantity associated with any call, such that:

- The routine's precondition implies that the variant is non-negative.
- If an execution of the routine starts with a value v for the variant, the value v' of the variant for any recursive call satisfies $0 \leq v' < v$.

The variant may involve the arguments of the routine, as well as other parts of its environment such as attributes of the current object or of other objects. In the examples just reviewed:

- For *Hanoi* (n, \dots), the variant is n .
- For *has*, *height*, *print_all* and other recursive traversals of binary trees, the variant is *node_height*, the longest length of a path from the current node to a leaf.
- For *animate_rest*, the variant is, as for the corresponding *loop*, $\text{Line8.count} - \text{Line8.index} + 1$. ← Page 166.

There's no special syntax for recursion variants, but we'll use a comment of the following form, here for *hanoi*:

```
-- variant  $n$ 
```

Boutique cases of recursion

The well-formedness rules seem so reasonable that we might think they are necessary, and not just sufficient, to make a recursive definition meaningful. That's indeed the case with the first two properties:

- **R1**: if all branches of a definition are recursive, it can't ever yield any instance we don't already know. If what's being defined is a recursive routine, its execution will not terminate, except possibly through a crash following memory exhaustion.
- **R2**: if a recursive branch applies to the original context, it can't ever yield an instance we don't already know. For a recursive routine — say $p(x: T)$ with a branch that calls $p(x)$ for the same x with nothing else changed — this means that the branch, if taken, would lead to non-termination. For other recursive definitions, it means the branch is useless.

The story is different for **R3**, at least if we take this rule to mean that there is a clearly visible recursion variant, such as the argument n for *Hanoi*. Some recursive routines which do terminate violate this property. Let's see two examples. They have no practical application, but highlight general properties of which we must be aware.

McCarthy's 91 function was devised by John McCarthy, a professor at Stanford University, designer of the Lisp programming language (where recursion plays a prominent role) and one of the creators of Artificial Intelligence. We may write it as follows:

```

mc_carthy (n: INTEGER): INTEGER is
  -- McCarthy's 91 function
  do
    if n > 100 then
      Result := n - 10
    else
      Result := mc_carthy (mc_carthy (n + 11))
    end
  end
end

```

The value for $n > 100$ is $n - 10$, but it's far less obvious — from a computation shrouded in two nested recursive calls — that for any integer up to 99, including negative values, the result will be 91, explaining the function's name. The computation indeed terminates on every possible integer value. Yet it doesn't have any obvious variant; *mc_carthy (mc_carthy (n + 11))* actually uses as argument of the innermost recursive call a *higher* value than the original.

Here is another example, also a mathematical oddity:

```

bizarre (n: INTEGER): INTEGER is
  -- A function that can yield only a 1
  require
    positive: n >= 1
  do
    if n = 1 then
      Result := 1
    elseif even (n) then
      Result := bizarre (n // 2)
    else
      -- i.e. for n odd and n > 1
      Result := bizarre ((3 * n + 1) // 2)
    end
  end
end

```

This uses the operator `//` for integer division, rounded down ($5 // 2$ and $4 // 2$ are both 2), and a boolean expression *even (n)* to denote whether n is an even integer; *even (n)* could also be expressed as $n \ \backslash\! \! \backslash \ 2 = 0$, using the integer remainder operator `\`. Note that the two cases of a `//` division in the algorithm apply to even numbers, so they are exact.

n / 2, using the other division operator /, would give a REAL result; for example $5 / 2$ is 2.5.

What's clear about this function is that if it gives any result at all (for a non-negative integer argument) that result can only be 1, the value produced by the sole non-recursive branch. What's not so clear is that it will give this result — that is to say, terminate — for *any* argument to which it is applied. This indeed seems to be the case; if you write the program, and try it on sample values, including large ones, you'll be surprised to see how fast it converges. Yet there is no obvious recursion variant; the new argument in the second recursive branch, $(3 * n + 1) // 2$, is indeed larger than n !

These are boutique examples, but we must take their existence into account in any general understanding of recursion. They mean that some recursive definitions exist that do *not* satisfy the seemingly reasonable methodological rules discussed above — and still yield well-defined results.

Note that such examples, if they terminate for every possible argument, do have a variant: since for any execution of the routine the number of remaining recursive calls is entirely determined by the program's state at the time of the call, it's a function of the state, and can serve as a variant. Rather, it *could* serve as a variant if we knew how to express it. If we don't, its theoretical existence doesn't help us much.

You will have noted that it's not possible to determine automatically — through compilers, or other program analysis tools — whether a routine has a recursive variant, even less to determine such a variant automatically: that would mean that we can solve the [Halting Problem](#).

← [“AN APPLICATION: PROVING THE UNDECIDABILITY OF THE HALTING PROBLEM”](#), page

In practice we will just dismiss such examples and limit ourselves to recursive definitions that possess the above properties, guaranteeing that they are safe. In particular, whenever you write a recursive routine, you must always — as in the examples of the rest of this chapter — explicitly list a recursive variant.

Keeping definitions non-creative

Even with well-formedness rules and recursion variants, we're not yet off the hook in our attempts to use recursion and still sleep at night. The problem is that a recursive “definition” is not a definition in the usual sense because it can be **creative**.

An *axiom* in mathematics is creative: it tells us something that we can't deduce without it, for example (in the standard axioms for integers) that $n < n'$ holds for any integer n , where n' is the next integer. The basic *laws* of natural sciences are also creative, for example the rule that no object can travel faster than the speed of light.

Theorems in mathematics, and specific results in physics, are not creative: they state properties that can be deduced from the axioms or laws. They are interesting on their own, and may start us on the path to new theorems; but they do not add any assumptions, only consequences of previous assumptions.

A definition too should be non-creative. It gives a new name for an object of our world, but all statements we can express with the definition could be expressed without it. We don't *want* to express them without it — otherwise we wouldn't have introduced the definition — but we trust that in principle we could. If I say

Define x^2 , for any x , as $x * x$

I haven't added anything to mathematics; I am just allowing myself to use the new notation e^2 , for any expression e , in lieu of the multiplication. Any property that can be proved using the new form could also be proved — if more clumsily — using the form that serves to define it.

The symbol \triangleq , which we have taken to mean “is defined as”, assumes this principle of non-creativity of definitions. But now consider a recursive definition, of the form

$f \triangleq \textit{some_expression}$ [D1]

where *some_expression* involves f . It doesn't satisfy the principle any more! If it did we could replace any occurrence of f by *some_expression*; this involves f itself, so we would have to do it again, and so on ad infinitum. We haven't really defined anything.

Until we have solved this issue — by finding a convincing, non-creative meaning for “definitions” such as [1] — we must be careful in our terminology. We'll reserve the \triangleq symbol for non-recursive definitions; a property such as [1] will be expressed as an equality

$f = \textit{some_expression}$ [D2]

which simply states a property of the left- and right-hand sides. (We may also view it as an **equation**, of which f must be a solution.) To be absolutely safe, in talking about recursive “definitions” we'll quarantine the second word in quotes.

The bottom-up view of recursive definitions

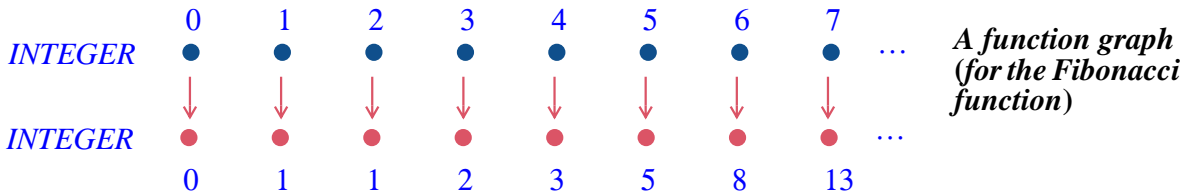
To sanitize recursive definitions and bring them out of the quarantined area, it is useful to take a **bottom-up** view of recursive routines and, more generally, recursive definitions. I hope this will remove any remaining feeling of dizziness that you may still experience when seeing concepts or routines defined — apparently — in terms of themselves.

In a recursive definition, the recursive branches are written in a *top-down* way, defining the value of a concept in terms of the value of the same concept for a “smaller” context — smaller in the sense of the variant. For example, *Fibonacci* for n is expressed in terms of *Fibonacci* for $n - 1$ and $n - 2$; the moves of *Hanoi* for n are expressed in terms of those for $n - 1$; and the syntax for *Instruction* involves a **Conditional** that contains a smaller *Instruction*.

The bottom-up view is a different interpretation of the same definition, treating it the other way around: as a mechanism that, from *known* values, gives new ones. Here is how it works, first on the example of a function. For any function f we may build the **graph of the function**: the set of pairs $[x, f(x)]$ for every applicable x . The graph of the Fibonacci function is the set

$$F \triangleq \{[0, 0], [1, 1], [2, 1], [3, 2], [4, 3], [5, 5], [6, 8], [7, 13] \dots\} \quad \text{[B1]}$$

consisting of all pairs $[n, \textit{Fibonacci}(n)]$ for all non-negative integers n . This graph contains all information about the function. You may prefer to think of it in the following visual representation:



The two row lists possible arguments to the function; for each of them, the bottom rows gives the corresponding *fibonacci* number.

To give the function a recursive “definition” is to say that its graph F — as a set of pairs — satisfies a certain property

$$F = h(F) \quad \text{[B2]}$$

for a certain function h applicable to such sets of pairs. This is like an equation that F must satisfy, and is known as a **fixpoint equation**.

For example to “define” the Fibonacci function recursively as:

$$\begin{aligned} \textit{fib}(0) &= 0 \\ \textit{fib}(1) &= 1 \\ \textit{fib}(i) &= \textit{fib}(i-1) + \textit{fib}(i-2) \quad \text{-- For } i > 1 \end{aligned}$$

is to state that its graph F — the above set of pairs [1] — satisfies the fixpoint equation $F = h(F)$ [2] where h is a function that, given such a set of pairs, yields a new one containing the following pairs:

G1 • $[0, 0]$ -- The pair for $n = 0$: $[0, fb(0)]$

G2 • $[0, 1]$ -- The pair for $n = 0$: $[1, fb(1)]$

G3 • Every pair already in F .

G4 • Every pair of the form $[i, a + b]$ for some i such that F contains both a pair of the form $[i - 1, a]$ and another of the form $[i - 2, b]$.

We can use this view to give any recursive “definition” a clear meaning, free of any recursive mystery. Let’s start from the function graph F_0 that is empty (it contains no pair). Next we define

$$F_1 \triangleq h(F_0)$$

meaning, since **G3** and **G4** are not applicable in this case (as F_0 has no pair), that F_1 is simply $\{[0, 0], [1, 1]\}$, with the two pairs given by **G1** and **G2**. Next we apply h once more to get

$$F_2 \triangleq h(F_1)$$

here **G1** and **G2** give us nothing new, since the pairs $[0, 0]$ and $[1, 1]$ are already in F_1 , but **G4**, applied to these two pairs from F_1 , adds to F_2 the pair $[2, 1]$. Continuing like this, we define a sequence of graphs: F_0 is empty, and each F_{i+1} for $i > 0$, is defined as $h(F_i)$. Now consider the infinite union F of all the F_i for every natural integer i : $F_0 \cup F_1 \cup F_2 \cup \dots$, more concisely written

$$\bigcup_{i \in \mathbf{N}} F_i$$

where \mathbf{N} is the set of natural integers. It is easy to see that this F satisfies the property $F = h(F)$ [2]. This is the non-recursive interpretation we give to the recursive “definition” of Fibonacci.

In the general case, a fixpoint equation of the form [2] on function graphs,, stating that F must be equal to $h(F)$, admits as a solution the function graph

$$F \triangleq \bigcup_{i \in \mathbf{N}} F_i$$

where F_i is a sequence of function graphs defined as above:

$$F_0 \triangleq \{ \} \quad \text{-- Empty set of pairs}$$

$$F_i \triangleq h(F_{i-1}) \quad \text{-- For } i > 0$$

The empty set can, of course, be written also as \emptyset . The notation $\{ \}$ emphasizes that it’s a set of pairs.

This fixpoint approach is the basis of the bottom-up interpretation of recursive computations. It removes the apparent mystery from these definitions because it doesn't any more involve defining anything "in terms of itself": it simply views a recursive definition as a fixpoint equation, and admits a solution obtained as the union (similar to the "limit" of a sequence in mathematical analysis) of a sequence of function graphs.

This immediately justifies the requirement that any useful recursive definition must have a non-recursive branch: if not, the sequence, which starts with the empty set of pairs $F_0 = \{ \}$, never gets any more pair, because all the cases in the definition of h are like G3 and G4 for Fibonacci, giving new pairs deduced from existing ones, but there are no pairs to begin with.

← [RI, page 385](#).

This technique reduces recursive "definitions", with all the doubts they raise as to whether they define anything at all, to the well-known, traditional notion of defining a sequence by induction.

The Fibonacci function provided a good example of this, but it's not in itself exciting since the usual definition of the function, in mathematics textbooks, is already by induction; it's only computer scientists who look at it in a recursive way. What we saw is that we can treat its recursive "definition" as an inductive definition — a good old definition, without the quotes — of the function's graph. This didn't teach us anything about the function itself, other than a new viewpoint. Let's see what we can learn about a couple of our other examples.

Bottom-up interpretation of a construct definition

Understood in a bottom-up spirit, the recursive definition of "type" has a clear meaning. As you will remember, it said that a type is either:

← "[Definition: Class type](#)", page 247.

T1 • A non-generic class, such as *INTEGER* or *METRO_STATION*.

T2 • A generic derivation, of the form $C [T]$, where C is a generic class and T is a type.

T1 is the non-recursive case. The bottom-up perspective enables us to understand the definition as building the set of types as a succession of layers. Limiting ourselves for simplicity to at most one generic parameter:

- Layer L_0 has all the types defined by non-generic classes: *INTEGER*, *METRO_STATION* and so on.
- Layer L_1 has all the types of the form $C [X]$, where C is a generic class and X is at level L_0 : *LIST [INTEGER]*, *ARRAY [METRO_STATION]* etc.
- More generally, layer L_n for any $n > 0$, has all the types of the form $C [X]$, where X is at level L_i for $i < n$.

This way we get all possible types, generically derived or not.

The towers, bottom-up

Consider the Tower of Hanoi solution from a bottom-up perspective. We may understand the routine as recursively defining a sequence of moves. Let's denote such a sequence — move a disk from the top of needle *A* to *B*, then one from *C* to *A* and so on — as $\langle A \rightarrow B, C \rightarrow A, \dots \rangle$. The empty sequence of moves will be $\langle \rangle$ and the concatenation of sequences will use a simple “+”, so that $\langle A \rightarrow B, C \rightarrow A \rangle + \langle B \rightarrow A \rangle$ is $\langle A \rightarrow B, C \rightarrow A, B \rightarrow A \rangle$. Then we may express the recursive solution to the Towers of Hanoi problem as a function *han* with four arguments (an integer and three needles), yielding sequences of moves, and satisfying the fixpoint equation

$$\begin{aligned} \text{han}(n, s, t, o) = & \\ & \langle \rangle \quad \text{-- If } n = 0 \quad \text{[N1]} \\ & \text{han}(n-1, s, o, t) + \langle s \rightarrow t \rangle + \text{han}(n-1, o, t, s) \quad \text{-- If } n > 0 \quad \text{[N2]} \end{aligned}$$

defined only when the values of *s*, *t*, *o* (short for *source*, *target*, *other*) are different — we'll take them as before to range over 'A', 'B', 'C' — and *n* is positive.

The bottom-up construction of the function that solves this equation is simple. [1] lets us initialize the function's graph to all pairs for $n = 0$, each of the form

$$[(0, s, t, o), \langle \rangle]$$

for *s*, *t*, *o* ranging over all permutations of 'A', 'B', 'C'. Let's call H_0 this first part of the graph, made of six pairs.

Now we may use [2] to obtain the next part H_1 , containing all the values for $n = 1$; they are all of the form

$$[(1, s, t, o), \langle s \rightarrow t \rangle]$$

since for any sequence *x* the concatenation $x + \langle \rangle$ is *x* itself. The next iteration of [2] gives us H_2 , whose pairs are of the form

$$[(2, s, t, o), fl + \langle s \rightarrow t \rangle + gl]$$

for all *s*, *t*, *o* such that H_1 contains both a pair of the form $[(1, s, o, t), fl]$ and one of the form $[(1, o, t, s), gl]$.

Iterating again will give us H_3 and subsequent elements of the graph. The complete graph — infinite of course, since it includes pairs for all possible values of n — is the set of all pairs in all elements of the sequence, $\bigcup_{i \in \mathbb{N}} H_i$.

Here I strongly suggest that you get a concrete grasp of the bottom-up view of recursive computation by writing a program that actually builds the graph:

Programming time:
Producing the graph of a function

Write a program (not using recursion) that produces successive elements $H_0, H_1, H_2 \dots$ of the function graph for the recursive Hanoi solution.

→ *Details in exercise 16-E.6, page 406.*

A related exercise asks you to determine (without programming) the mathematical properties of the graph.

→ *16-E.5, page 405.*

Another important exercise directs you to apply a similar analysis to binary tree traversals. You'll have to devise a model for representing the solution, similar to the one we've used here; instead of sequences of moves you'll simply use sequences of nodes.

→ *16-E.7, page 406.*

Grammars as recursively defined functions

The bottom-up view is particularly intuitive for a recursive grammar, as in our small example:

Instruction \triangleq **ast** | Conditional
Conditional \triangleq **ifc** Instruction **end**

← *Actual version on page 361.*

distilled even further here: **ifc** represents “**if Condition then**” and **ast** represents **Assignment**, both treated as terminals for this discussion.

It's easy to see how to generate successive sentences of the language by interpreting these productions in a bottom-up, fixpoint-equation style:

```
ast
ifc ast end
ifc ifc ast end end
ifc ifc ifc ast end end end
```

and so on. You can also look again, in light of the notion of bottom-up recursive computation, at the earlier discussion of the little **Game** language.

← *“Recursive grammars”, page 338.*

It is possible to generalize this approach to arbitrary grammars by taking a matrix view of a BNF description.

→ *Exercise 16-E.9, page 406.*

16.6 CONTRACTS FOR RECURSIVE ROUTINES

We have learned to equip our classes and their features with **contracts** stating their correctness properties: routine preconditions, routine postconditions, class invariants; the same concerns applied to algorithms gave us loop variants and loop invariants. How does recursion affect the picture?

We have already seen the notion of **recursion variant**. If a routine is recursive directly or indirectly, you should include a mention of its variant. As noted, we don't have specific language syntax for this but add a clause

← “*Touch of Methodology: Recursion Variant*”, page 386.

```
-- variant: integer_expression
```

to the routine's header comment.

A recursive routine may have a precondition and postcondition like any other other routine. Because ensuring a precondition is always the responsibility of the caller, and here the routine is its own caller, the novelty is that you must ensure that all calls within the routine (or, for indirect recursion, in associated routines) satisfy the precondition.

Here is the Towers of Hanoi routine with more complete contracts; the new clauses, all expressed as comments, are highlighted.

← The original was on page 367.

```
hanoi (n: INTEGER; source, target, other: CHARACTER) is
  -- Transfer n disks from source to target, using other as intermediate
  -- storage, according to rules of Tower of Hanoi puzzle
  -- variant: n
  -- invariant: disks on each needle are piled in decreasing size
  require
    non_negative: n >= 0
    different1: source /= target
    different2: target /= other
    different3: source /= other
  -- source has n disks; any disks on target and other are all
  -- larger than all the disks on source.
  do
    if n > 0 then
      hanoi (n-1, source, other, target)
      move (source, target)
      hanoi (n-1, other, target, source)
    end
  ensure
    -- Disks previously on source are now on target, in same order,
    -- on top of those previously there if any; other is as before
  end
```

A recursive routine often has a **recursion invariant**: a set of properties that must hold both before and after each execution. In the absence of a specific language mechanism they will just appear twice, in the precondition as well as the postcondition; for clarity we may also, as here, include them in the header comment under the form

```
-- invariant: integer_expression
```

Since this is not a language construct we will use the following convention:

- If the recursion invariant is just a comment, as in this example, we will not repeat it in the precondition and postcondition; here this means omitting from the precondition and postcondition that any disks on the affected needles are in decreasing size.
- Any recursion invariant clause that's formal (a boolean expression) should be included in the precondition and postcondition, since there is no other way to express it formally.

16.7 THE IMPLEMENTATION OF RECURSIVE ROUTINES

Recursive programming works well in certain problem areas, as illustrated by the examples in this chapter. When recursion facilitates your job you shouldn't hesitate to use it, since in modern programming languages you can take recursion for granted.

Since there's usually no direct support for recursion in machine code, compilers for high-level languages must be able to map a recursively expressed algorithm into a non-recursive one. The applicable techniques are obviously important for compiler writers, but even if you don't expect to become one it's useful to know the basic ideas, both as a way to gain further insight into recursion (complementing the various perspectives opened by previous sections) and to understand the potential performance cost of using recursive algorithms.

We'll look at a simple recursive routine and ask ourselves how, if the language did *not* permit recursion, we would achieve its aims.

A recursive scheme

Consider a routine r that calls itself:

```

r (x: T) is
  do
    code_before
    r (y)
    code_after
  end

```

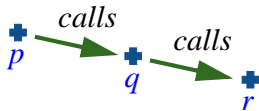
What does it mean — looking at things top-down again — to execute the recursive call?

The recursiveness of this scheme implies that neither the beginning of the routine’s code nor its end are just what they appear to be:

- When *code_before* executes, this is not necessarily the beginning of a call $a.r(y)$ executed by a client: it could result from an instance of r calling itself recursively.
- When *code_after* terminates, it’s not necessarily the end of the r story: it may simply be the termination of one recursively called instance; execution should resume for the last instance started and not terminated.

Routines and their execution instances

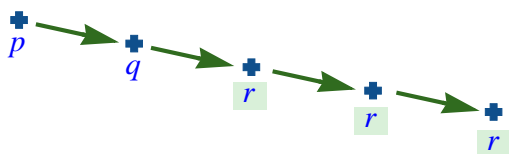
The key novelty in the last observation is the concept of **instance** of a routine. We know that classes have instances (the “objects” of object-oriented program execution) but we haven’t yet thought of routines in a similar way.



*A call chain,
without recursion*

Execution of a program is, at any point during execution, characterized by a **call chain** as pictured above: the root procedure p has called q which has called r ... When an execution of a routine in the chain, say r , terminates, the suspended execution of the calling routine, here q , resumes just after the place where it had called p . In the absence of recursion, any procedure has at most one instance active at any time.

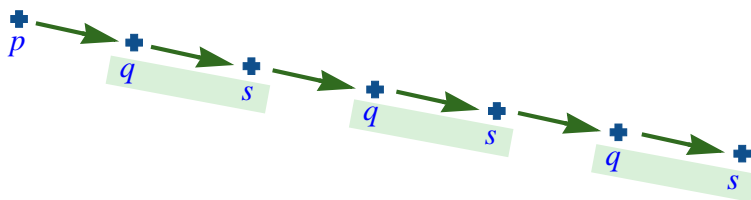
With recursion, the call chain may include two or more instances (also called **activations**) of the same routine. Under *direct* recursion they will be contiguous:



Call chain with direct recursion

For example a call $hanoi(2, s, t, o)$ immediately starts a call $hanoi(1, s, o, t)$ which starts a call $hanoi(0, s, t, o)$; at that stage we have three instances of the procedure in the call chain.

A similar situation arises with *indirect* recursion:



Call chain with indirect recursion

Preserving and restoring the context

All instances of a routine share their program code; what distinguishes them is their execution *context*. We've seen that in a meaningful case of recursion the context of every call must differ by at least one element. The context elements characterizing a routine instance (rather than object states) are:

← [R2, page 385](#).

- The values of the actual routine arguments, if any, for the particular call.
- The values of the local variables, if any.
- The location of the call in the text of the calling routine, defining where execution should continue once the call completes.

A data structure representing such a routine execution context is called an **activation record**.

Assume for the moment a programming language that does *not* support recursion. Since at any time during execution there's at most one active instance of any routine, the compiler-generated program can use a single activation record for each routine. This is known as **static allocation**, meaning that the memory for all activation records can be allocated once and for all at the beginning of execution.

With recursion each activation of the routine needs its own context. This leaves two possibilities for implementation:

- I1 • We can resort to *dynamic allocation*: whenever a routine instance starts, create a fresh activation record to hold the routine's context. Use this activation record whenever the routine execution needs access to an argument or local variable, and to determine where in the caller to transfer execution on return; resuming the caller's execution will imply going back to its activation record.
- I2 • To save space, we may note that the reason for keeping context information in an activation record is to be able to *restore* it after when an execution resumes after a recursive call. An alternative to saving that information is to *recompute* it. This is possible when the change performed by the recursive call is **invertible**. The recursive calls in procedure *hanoi* (n, \dots) are of the form *hanoi* ($n - 1, \dots$); rather than storing the value of n into an activation record, creating a new one with $n - 1$ at the corresponding position, then restoring the value on return, we may use a single location for n in all recursive instances, as with static allocation: at call time, we'll decrease the value by one; at return time, we'll *increase* the value by one.

The two techniques are not exclusive: you can save space by using using [I2](#) for values whose transformation (such as replacing n by $n - 1$) admits an easily implemented inverse, but keep an activation record for the rest of the context. The decision may involve a space-time tradeoff if the reverse transformation takes more time than the $n := n + 1$ of our example.

Stacks

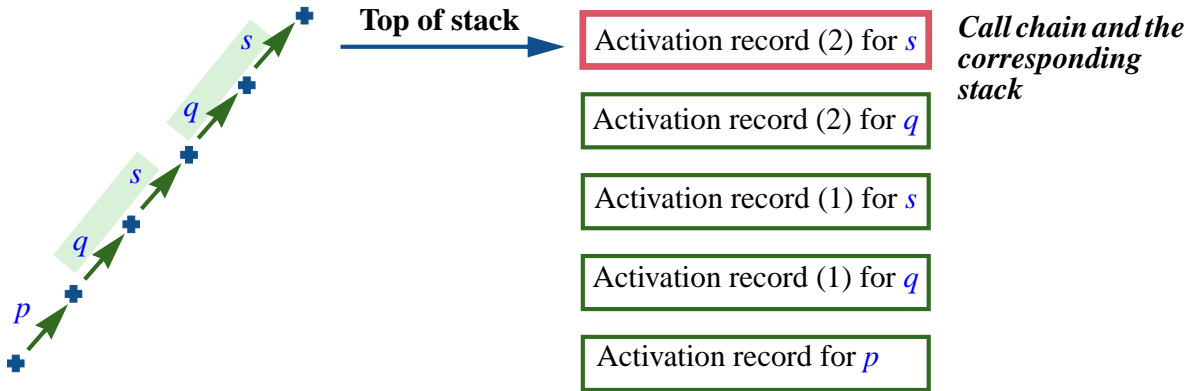
Of the two strategies for handling routine contexts let's look first at [I1](#), the explicit use of activation records.

Like activation records, *objects* are created dynamically, as a result of **create** instructions. The program memory area devoted to to dynamically allocated objects is known as the **heap**. But for activation records of routines we don't need to use the heap since the patterns of activation and deactivation are simple and predictable:

- A call to a routine requires a new activation record.
- On returning from that call, we may forget this activation record (it will never be useful again, since any new call will need its own values), and we must restore the caller's activation record.

This is a *last-in, first-out* pattern for which we have a ready-made data structure: stacks. The stack of activation records will reflect the call chain, pictured here going up:

← “[STACKS](#)”, 10.9, page 293.



Using a stack of activation records we can provide a non-recursive version of a recursive routine:

- Instead of a recursive call: create a new activation record; initialize it with the value of the call’s arguments and the position of the call; push it on the stack; and branch back (**goto**) to the beginning of the routine’s code.
- Instead of a return: return only if the stack is empty (no suspended call is pending); otherwise, restore the arguments and local variables from the activation record at top of the stack, pop the stack, and branch to the appropriate instruction based on the call position information found in the activation record.

Note that both steps involve **goto** instructions. That’s OK if we are talking about the machine code to be generated by a compiler, but if it is a manual simulation of recursion in a high-level language we have learned to avoid the **goto** and in fact Eiffel doesn’t have such an instruction. We’ll have to use **gotos** temporarily and then replace by the appropriate control structures.

← “[The goto instruction and flowcharts](#)”, page 181.

Let’s see how this will work for the body of *hanoi* with its two recursive calls. We’ll use a stack of tuples:

activations: STACK [HANOI_ACTIVATION_RECORD]

with a small auxiliary class *HANOI_ACTIVATION_RECORD*:

```

class HANOI_ACTIVATION_RECORD create
  make
feature
  make (c: INTEGER; n: INTEGER; s, t, o: CHARACTER) is
    -- Initialize from the values given.
  do
    call := c; count := n; source := s; target := t; other := o
  end
  count: INTEGER
    -- Number of disks
  call: INTEGER
    -- Identifies a recursive call: 1 for the first, 2 for the second
  source, target, other: CHARACTER
    -- Needles
end

```

(Instead of a full-fledged class we could also just use tuples.) An instance of the class represents the context of a call: number of disks being moved (*count*), the three needles in the order used by the call, and *call* telling us whether this execution, if coming from a recursive call, came from the first or second call in

```

hanoi (n: INTEGER; source, target, other: CHARACTER) is
  do
    if n > 0 then
      hanoi (n-1, source, other, target)
      move (source, target)
      hanoi (n-1, other, target, source)
    end
  end
end

```

Based on the preceding discussions, we can use the stack of activation records to provide a non-recursive version of the procedure, relying on **gotos**: This is the kind of code that a compiler would generate for a recursive routine, in the absence of any “optimization” as described next. (Since machine language usually does not directly support **if ... then ... else ... end** instructions, the code will represent them through conditional branches **test ... goto ...** as seen in an earlier chapter.) The whole code can be expressed in Eiffel through techniques of **goto** elimination; this is the subject of an exercise.

← “The goto instruction and flowcharts”,
 ← 7-E.5, page 196.

The body of *hanoi_derecursified* derives from *hanoi* through systematic application of recursion elimination techniques:

- D1 • For every argument, introduce a local variable. (You'll have noted the naming convention of the exampleL *l_source* for *source* and so on.) Assign the value of the argument to the local variable on entry, then work exclusively on the local variable. This is necessary because a routine may not, for obvious reason, change the value of its own arguments (*source := some_new_value* is invalid.)
- D2 • Give a label, here *start*, to the routine's original first instruction (past the local variable initializations added by D1).
- D3 • Give a label, here *after_1* and *after_2*, to the instructions immediately following each recursive call.
- D4 • Replace each recursive call by instructions which: push on the stack an activation record containing the values of local variables and the identification of the call (here **1** or **2**); modify the local variables representing arguments to reflect the values of the recursive call's actual arguments (here the recursive call replaces *n* by *n - 1* and swaps the values of *other* and *target*, using the local variable *swap* for that purpose); and branch to the first instruction.
- D5 • Add to the end of the routine instructions which terminate the routines' execution only if the stack is empty, and otherwise: restore the values of all local variables from the activation record at the top of the stack; also from that record, obtain the call identification; branch to the appropriate post-recursive-call label among those set in D3.

Taking advantage of invertible functions

===== STOPPED HERE =====

16.8 GENERAL TREES

16.9 BACKTRACKING ALGORITHMS AND ALPHA-BETA SEARCH

16.10 APPENDIX: AVL TREES

16.11 APPENDIX: ITERATIVE HANOI

```

start:      l_n := l_n - 1
            swap := l_target ; l_target := l_other ; l_other := swap
            goto start

after_1:    move (l_source, l_target)

            l_n := l_n - 1
            swap := l_target ; l_target := l_other ; l_other := swap
            goto start

after_2:    if not activations.is_empty then
            top := activations.item -- Top of stack
            l_n := top.count ; l_source := top.source ;
            l_target := top.target ; l_other := top.other
            call := top.call ; activations.remove
            if call = 1 then
                goto after_1
            else
                goto after_2
            end
            end

end

end

-- Translation of routine return:
-- No else clause: the routine terminates when (l_source = l_target).
-- (and only when) the stack is empty.

wap

```

a

end

16.12 KEY CONCEPTS LEARNED IN THIS CHAPTER

New vocabulary

Activation	Activation record	Call chain
Direct recursion	Indirect recursion	Instance (of a routine)
Non-creative	Recursion	Recursive
Recursive definition		

16.13 FURTHER READING

The theory of *denotational semantics*, which provides a mathematical behind recursive functions and more generally

16-E EXERCISES

16-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

16-E.2 Too much recursion?

Is the definition of “recursive definition” a recursive definition?

← Page [359](#).

16-E.3 Binary search trees with repetitions

For every binary search tree routines in this chapter, rewrite the declaration (if needed) to permit multiple occurrences of a given *item* value in a tree as discussed after the initial definition.

← Page [377](#).

16-E.4 Non-recursive insertion

Write a version of *put* binary search trees using a loop rather than recursion. (**Hint**: you may use for inspiration the non-recursive version of the search function *has*.)

← Page [380](#).

16-E.5 Properties of a function graph

(This exercise requires a mathematical analysis, not a programming solution.) In the successive approximations H_i of the graph of the Towers of Hanoi function, assuming three needles ‘A’, ‘B’, ‘C’:

← “*The towers, bottom-up*”, page [393](#).

- 1 • What is the number of pairs in H_i ?
- 2 • Devise a formula

16-E.6 Programming a function graph bottom-up

- 1 • Devise a class of which every instance represents an arguments-result pair, of the form $[(n, s, t, o), <...>]$, for the the Towers of Hanoi function graph. ← *“The towers, bottom-up”, page 393.*
- 2 • Based on the preceding class, devise another to represent the function graph as a whole.
- 3 • From this class and the rules [1] and [2] defining the function graph in the bottom-up interpretation of recursion, write a program that produces the i -th approximation of the graph, H_i , for any i . The algorithm may use loops, but it may not use recursion.
- 4 • Use this program to print out sequences of moves (with source ‘A’ and target ‘B’) for a few values of i ; check that the results coincide with those of the recursive procedures.

16-E.7 Bottom-up view of binary tree algorithms

Consider a recursive algorithm for binary tree traversal; you may choose preorder, inorder or postorder.

- 1 • Taking inspiration from the bottom-up analysis of the Towers of Hanoi solution, devise a model to interpret the traversal as a function returning a sequence of nodes. ← *“The towers, bottom-up”, page 393.*
- 2 • Write a recursive “definition” of this function.
- 3 • Express this “definition” as a fixpoint equation on the function graph, using T_i as the name of the graph for binary trees of height i .
- 4 • Use the definition to produce (either manually or by writing a small program) H_5 for the example binary tree, and the resulting traversal order. ← *From the figure on page 370.*

16-E.8 Transitive closure

[This exercise refers to a later chapter.] Restate the definition of transitive closure as a recursive definition. → *Page 417.*

16-E.9 Matrix algebra on BNF productions

(This exercise requires a basic knowledge of linear algebra.)

Consider a BNF production, such as the small example used in this chapter, or more extensive ones from earlier chapters, involving only Concatenation and Choice productions (no Repetition, which can however be replaced by combinations of the other two).

- 1 • Treating concatenation of tokens as “multiplication” and alternative choices as “addition”, show that it’s possible to express the grammar as a matrix equation $X = A * X + B$, where X is the vector of nonterminals, A is a matrix of terminals and nonterminals, and B a vector.
- 2 • Discuss ways of solving this equation by following the model discussed for fixpoint equations.

17

An elegant algorithm family: Topological Sort

One of the pleasures of learning computer science is to discover beautiful algorithms. In this chapter we explore an algorithm scheme with many complementary benefits: it is useful in many practical situations; it has a simple mathematical basis; it is particularly elegant; and it illustrates problem-solving techniques that you will find applicable in many other contexts.

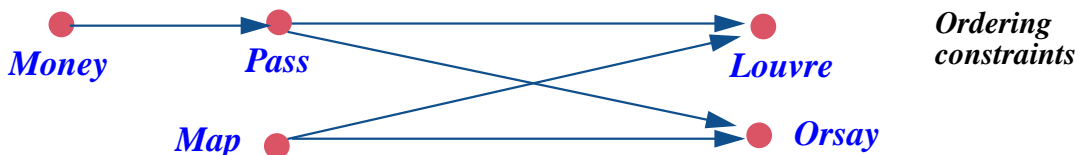
I won't throw a ready-made answer at you; instead we'll develop the solution step by step from the description of the problem, starting with a mathematical analysis and continuing with a search for data structures ensuring both correctness and efficiency. We will not just devise an algorithm but strive for a complete, properly *engineered* solution that can satisfy practical needs. At the end of the chapter, we'll draw the lessons of this example for both algorithm development and general software engineering.

17.1 THE PROBLEM

Today is for culture: you want to visit the Louvre and the Orsay museum, in any order. Before visiting either you must get a map; you must also get a metro pass because your old one expired yesterday, but you can't get a pass until you've gone to the bank or an Automatic Teller Machine to get some money. We may express these constraints as

$[Map, Louvre], [Map, Orsay], [Pass, Louvre], [Pass, Orsay], [Money, Pass]$

where $[x, y]$ means “ x must occur before y ”; or we may represent them graphically



A **topological sort** of such a set of elements governed by ordering constraints is an enumeration of all the elements in an order that respects the constraints. Possible topological sorts in this example include

There are two more possibilities

Money, Pass, Map, Louvre, Orsay
Money, Pass, Louvre, Map, Orsay
Money, Map, Pass, Louvre, Orsay

but *Pass, Money, Map, Louvre, Orsay*, for example, would be incorrect since it violates the constraint [*Money, Pass*].

A topological sort problem may have:

- Several solutions, as here.
- Exactly one solution.
- No solution, as will be the case if — and only if — the constraints include a *cycle*: a set of constraints of the form $[e_1, e_2], [e_2, e_3], \dots, [e_n, e_1]$ for some $n \geq 1$. If we add [*Orsay, Money*] to the example, creating such a cycle, there can't be any solution since the constraints require both that *Money* occur before *Orsay* and the reverse.

If there's more than one solution, the problem is to produce *one* of them. In practice, there's often a cost function associated with any solution; then the goal will be to produce the solution with minimal cost. We will see where, in the algorithm, we can apply this criterion to choose between alternative solutions. Another variant of the problem would be to produce *all* solutions.

Examples

The topological sort problem arises whenever we want to order a number of elements in conformance to some ordering constraints. This is a frequent problem; here are some examples.

- In a graphical display, consider a set of rectangles that partially overlap. Some are “on top” of others, as illustrated. You need an algorithm that will display the rectangles in a certain order respecting these constraints, so that in the end the figure appears as intended. This is a topological sort problem. In the illustrated example, the constraints are [*B, A*], [*D, A*], [*D, C*], [*B, D*], [*E, C*]; a possible solution is the order **B D E A C**.

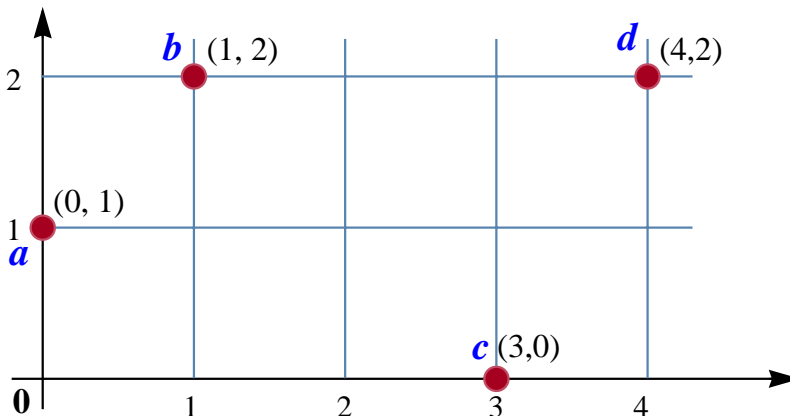


Rectangles with overlap constraints

- When an industrial installation such as a power plant or an airplane undergoes *maintenance*, the schedule is determined from a set of tasks to be performed and a set of ordering constraints between them; for example structural work on an element must come before repainting it. A topological sort yields a schedule of tasks compatible with these constraints.
- Another application occurs in *project management*, especially software project management. If the problem domain is technical, the project should produce and maintain a *glossary* of the technical terms involved. (Misunderstandings between application area experts and software developers are a prime source of errors and deficiencies in software systems.) The definition of any of these terms may involve other terms that have their own entries. The entries might appear in alphabetical order, as in a dictionary, but it may also be useful to have a version of the glossary that can be read in sequence, with the definition of any term appearing before any definition that uses the term. Producing such a list is a topological sort problem.
- You might want to see a list of the *features in an Eiffel class* that shows the features not in the order listed (grouped, by default, into feature categories) but in one that facilitates a sequential reading by guaranteeing that no call to a feature occurs before the feature's declaration.

Points in a plane

Another example provides a convenient visualization of the problem. Consider points in a plane:



A finite set of points

We introduce a relation \ll by stating that $p_1 \ll p_2$ holds for any two points p_1 of coordinates (x_1, y_1) and p_2 of coordinates (x_2, y_2) if they satisfy all of:

- $x_1 \leq x_2$
- $y_1 \leq y_2$
- $p_1 \neq p_2$ (the two points are not equal).

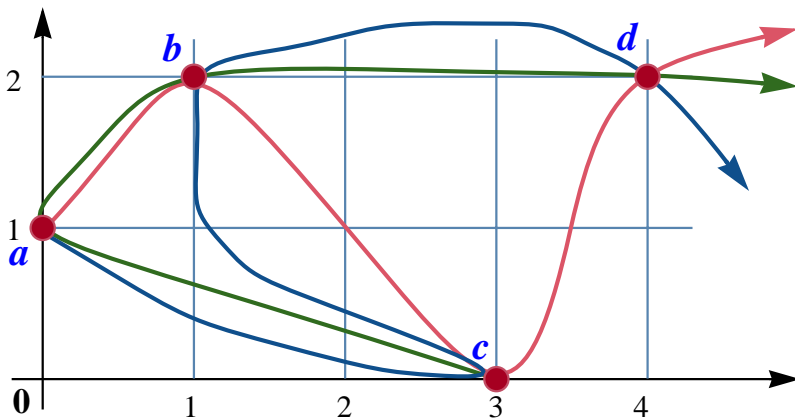
For the four points shown on the figure, the following hold:

$$a \ll b \quad a \ll d \quad b \ll d \quad c \ll d$$

A topological sort for this relation is any enumeration of the points which lists p before q for any two points such that $p \ll q$. For our four points there are three such enumerations:

$$\begin{array}{l} a, b, c, d \\ a, c, b, d \\ c, a, b, d \end{array}$$

which we may illustrate for the preceding figure:



*Three
topological
sorts of a set
of points*

On the other hand, the enumeration a, d, b, c is not compatible with the order relation \ll since the property $c \ll d$ requires c to appear before d .

17.2 THE BASIS FOR TOPOLOGICAL SORT

The problem discussed in this chapter has a terse mathematical formulation:

Definition: The topological sort problem

Given an acyclic relation r on a finite set, find a total order relation of which r is a subset.

This definition is made possible by simple mathematical notions — relations as sets, acyclic relation, order relation (total or not), which we'll review now.

Binary relations

Definition: Relation

A **relation** over a set A (short for *binary* relation) — is a set of pairs of the form $[x, y]$ where both elements of the pair, x and y , are members of A .

An example relation over the set $\{1, 2, 3\}$ is

$\{[1, 2], [1, 3], [2, 3]\}$

which we may call " $<$ " since it represents “less than” (meaning that it contains all the pairs $[x, y]$, with a and b both in $\{1, 2, 3\}$, such that x is less than y).

We may use relations to describe the earlier examples:

- A relation *on_top* over a set of rectangles, containing all rectangle pairs $[x, y]$ such that the display must show points of x rather than y in any area where they overlap.
- A relation *before*, the set of pairs $\{[Map, Louvre], [Map, Orsay], \dots\}$; it's a relation over the set $\{Money, Pass, Map, Louvre, Orsay\}$, containing all pairs $[x, y]$ for which we want to express that x must happen before y .
- A relation *used_in* over a set of glossary terms, containing all pairs $[x, y]$ such that the definition of the term y uses the term x .
- A relation *called_by* over the features of a class, containing all pairs $[x, y]$ such that the body of feature y contains a call to feature x .
- A relation \ll over points, the set of pairs $\{[a, b], [a, d], [b, d], [c, d]\}$.

Acyclic relations

Our examples so far are all *acyclic relations*, a notion defined as follows:

Definition: Acyclic relation

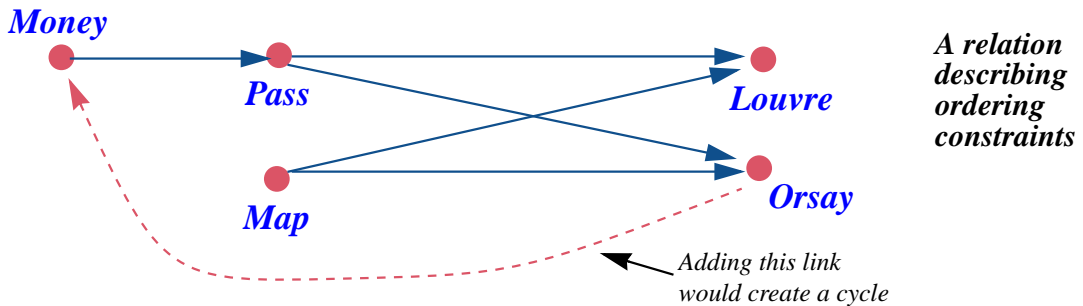
A relation is **acyclic** if it has no cycle.

with:

Definition: Cycle in a relation

A **cycle** for a relation r over a set A is a sequence x_1, \dots, x_m ($m \geq 1$) of elements of A such that all successive pairs $[x_i, x_{i+1}]$ for $1 \leq i < m$ belong to r , and $x_m = x_1$.

The relation *before* as given has no cycles:



Adding the pair $[Orsay, Money]$ would create a cycle: *Money, Pass, Orsay, Money*.

To succeed, topological sort requires an acyclic relation, although we'll look for an algorithm that can partially process constraints involving a cycle.

If the underlying set is finite, acyclic relations have an important property, crucial to the topological sort algorithm:

No-Predecessor theorem

For any acyclic relation p over a non-empty finite set A , there exists an element x of A with no predecessors for p .

relying on a notion of “predecessor”:

Definition: Predecessor

A **predecessor** of an element y for a relation r is an element x such that the pair $[x, y]$ belongs to r .

The proof of the No-Predecessor theorem is by contradiction. Assume the theorem doesn't hold; then every element in A has at least one predecessor. Let x_1 be some element in A (we may indeed find such an x_1 since the theorem assumes A to be non-empty). By the hypothesis x_1 has at least one predecessor; let's pick one and call it x_2 . By the same reason x_2 also has at least one predecessor, so we may again pick x_3 such that $[x_3, x_2]$ is in the relation. Continuing this way gives an infinite sequence such that $[x_{i+1}, x_i]$ belongs to the relation for every $i \geq 1$. Because A is a finite set, the sequence has to repeat elements: more precisely the elements x_1, x_2, \dots, x_{n+1} , where n is the number of elements of A , cannot all be different. In other words there must be integers i and j , with $1 \leq i < j \leq n+1$, such that $x_i = x_j$. But then x_j, x_{j-1}, \dots, x_i is a cycle for the relation, which is impossible.

This is a *constructive* proof, which we'll use directly in devising the topological sort algorithm: to produce an enumeration of the elements, the algorithm will pick, at every iteration, an element that has no predecessor in the remaining order relation.

The condition that A is finite is essential to the proof. The theorem doesn't apply to infinite sets; for example, the relation “less than or equal” on mathematical integers is acyclic, but every element has a predecessor.

Order relations

The idea of topological sort is to embed a given acyclic relation in a *total order* relation. To define this notion we must first consider plain *order* relations.

Definition: Order relation (strict, possibly partial)

A relation is an **order** relation if it satisfies the following properties for any elements x, y, z of the underlying set X :

- O1 • **Irreflexive**: the relation has no pair of the form $[x, x]$.
- O2 • **Transitive**: whenever the relation contains a pair $[x, y]$ and a pair $[y, z]$ (whose first element is the same as the second element of the first pair), it also contains the pair $[x, z]$.

Such an order relation is also:

O3 • Asymmetric: whenever it contains a pair $[x, y]$, it does *not* contain the pair $[y, x]$. (Proof: if it contained both, transitivity implies that it would also contain $[x, x]$, violating irreflexivity.)

The full name for order relations as defined above is: *strict and possibly partial* order relation. Our order relations (also the “total” ones seen next) are *strict*, in the same sense that “ $<$ ” denotes “*strictly* less than”. It’s also possible to work with the nonstrict versions, such as “ \leq ”, less than or equal.

→ See “[TERMINOLOGY NOTE: ORDER RELATIONS](#)”, 17.7, page 450. Also, do exercise 17-E.3, page 451 to explore the relationship between strict and non-strict versions.

The relation “ $<$ ” on $\{1, 2, 3\}$ (or any other set of integers) is an order relation. So is the relation \ll on points. Our other acyclic relations — *before* between tasks, *used_in* between dictionary entries, *called_by* between features — are irreflexive and asymmetric, but not necessarily transitive, so they are not order relations; we’ll see next how to obtain transitive versions.

Order relations vs acyclic relations

Order relations are closely connected with acyclic relations. In one direction the connection is quite clear:

Theorem: Acyclic and order relations (1)

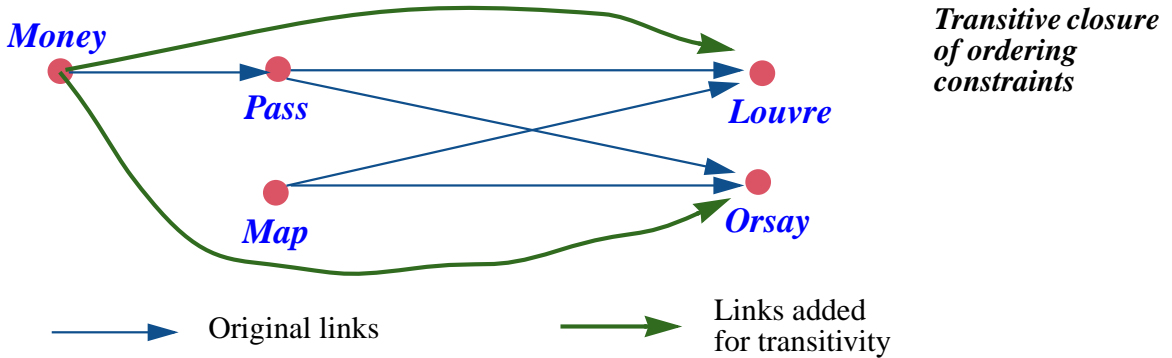
Any order relation (and more generally any subset of an order relation) is acyclic.

The proof is by contradiction. Assume a cycle $x_1, x_2, x_3, \dots, x_m$ where x_m is the same as x_1 . By transitivity (O2) this implies that $[x_1, x_1]$ is also in the relation; that’s impossible because of irreflexivity (O1).

This proof generalizes the earlier proof that asymmetry — the impossibility of having both pairs $[x, y]$ and $[y, x]$ (O3) — follows from O1 and O2. Such a case is indeed a cycle with just two elements. Similarly, the pair $[x, x]$, ruled out by irreflexivity, would be a cycle with just one element.

There’s also an interesting property the other way around: the *transitive closure* of an acyclic relation is an order relation. Informally, the transitive closure of a relation is a version of the relation made transitive by following the original relation’s links as many times as possible.

This can be illustrated on the relation *before* expressing ordering constraints between tasks. The relation is irreflexive, asymmetric and acyclic; it is not transitive since it contains the pairs *[Money, Pass]* and *[Pass, Louvre]* but not *[Money, Louvre]*. We can make such a relation transitive by adding all pairs of the form *[x, z]* for which the original includes both *[x, y]* and *[y, z]* for some *y*, repeatedly until there are no more pairs to be added. The result of this process is the transitive closure of the original relation. In the example it adds just two links:



For the relation *called_by* between features, the transitive closure is the relation that holds between *x* and *y* if *y* calls *x* directly or indirectly. For a relation *child* among persons, denoting the set of pairs *[x, y]* such that person *x* is a child of *y*, the transitive closure is the relation connecting any two persons *x* and *y* such that *y* is a descendant, direct or indirect, of *x*.

The transitive closure of a relation *r* is written *r⁺*, so we may state that *child⁺ = descendant*. Here is a precise definition:

Definition: Transitive closure of a relation

The **transitive closure** *r⁺* of a relation *r* over a set *A* is the relation containing all pairs of the form *[x₁, x_m]* for some sequence of elements *x₁, ... x_m* (*m* ≥ 1) such that all *[x_i, x_{i+1}]* pairs for 1 ≤ *i* < *m* belong to *r*.

←Exercise 16-E.8, page 406 requests a recursive variant of this definition.

Transitive closure gives us the other side of the relation between acyclic relations and order relations:

Theorem: Acyclic and order relations (2)

The transitive closure of any acyclic relation is an order relation.

Proof: the transitive closure of any relation r is obviously transitive, so all we have to show is that it's irreflexive for an acyclic r . Assume it isn't. This means there's an element x such that $[x, x]$ belongs to r^+ . By the definition of transitive closure there must be a sequence of elements x_1, \dots, x_m ($m \geq 1$) such that all $[x_i, x_{i+1}]$ pairs for $1 \leq i < m$ belong to r and that both x_1 and x_m are x . But this is a cycle for r , and hence impossible from the previous theorem.

This result shows that we may view an acyclic relation as the “germ” of an order relation. Taking its transitive closure gives us a true order relation. This corresponds to the intuition behind relations such as *before* between tasks. If the constraints specify that task *Money* must precede *Pass*, and also that *Pass* must precede *Louvre*, we naturally understand that x must precede z ; in other words, we instinctively take the transitive closure. But when it comes to preparing the input data for a scheduling program, or another program that will perform a topological sort, we'll want to list basic constraints only, not their full transitive closure. That's why topological sort can use an acyclic relation as its input. (Many presentations of topological sort start from an order relation, but that's more specific than required.)

Computing a transitive closure is a computationally expensive operation, but we don't need to perform it explicitly for topological sorting; we will just work from the acyclic relation.

Total orders

To describe the output of topological sorting we need a specialization of the notion of order relations: *total* order relations. For a finite set we may view a total order simply as an enumeration of the underlying set's elements, each appearing once, such as *Money, Pass, Map, Louvre, Orsay*; but the concept is more general:

← Briefly encountered in the study of recursion: “[Binary search trees](#)”, page 377.

Definition: total order relation (strict)

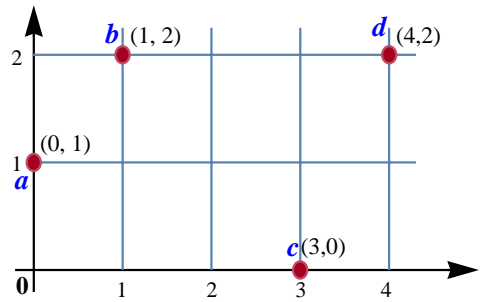
A total order is an order relation that additionally is:

O4 • **Total**: for any a and b , one of the following holds: $[a, b]$ is in the relation; $[b, a]$ is in the relation; $a = b$.

← Meaning a relation that's irreflexive (O1, page 415) and transitive (O2); it's asymmetric as a result (O3).

To understand condition O4, note that we know from asymmetry (O3) that at most one of the first two possibilities may hold, and from irreflexivity (O1) that the last possibility is exclusive of the other two. So *at most one* of the three may hold. What the new condition adds is that one of the three *does* hold.

The relation " $<$ " on integers is also total. But not every order is total. Our \ll relations on points in a plane is an order relation, as we have seen; but it is not total since this would mean that for any two different points p_1 and p_2 either $p_1 \ll p_2$ or $p_2 \ll p_1$ holds. That's not the case for the pair $[a, c]$ since neither $a \ll c$ nor $c \ll a$ holds. There is another counter-example, the pair $[b, c]$.



(Figure from page 411.)

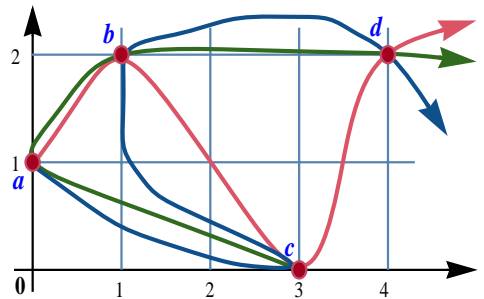
Many total orders exist on this set of four points; in fact, any enumeration of them — that is to say, any ordered list that includes each of them exactly once — yields a total order t , defined as follows: the pair $[p, q]$ is in t if and only if p appears before q in the enumeration. For example the enumeration $[a, b, c, d]$ defines the total order

$$\{[a, b], [a, c], [a, d], [b, c], [b, d], [c, d]\}$$
[1]

Conversely, any total order on a finite set defines a single enumeration.

→ The proof is exercise 17-E.3, page 451.

Such a total order is a topological sort of the original relation — in our example, the relation \ll — if and only if it is *compatible* with it, meaning that whenever $p \ll q$ the element p appears before q in the total order. We have seen that three total orders satisfy this requirement for the example: expressed as enumerations they are a, b, c, d (expressed in [1] as a set of pairs); a, c, b, d ; and c, a, b, d .



(Figure from page 412.)

What does “compatible” precisely mean? It’s actually very simple thanks to our definition of relations as sets of pairs. To say that a total order such as the enumeration a, b, c, d , is compatible with a given (acyclic) relation is simply to say that the set of pairs of that relation is a **subset** of the total order’s set of pairs: every pair in the order relation is also a pair of the total order relation. In our example the relation \ll is the set of pairs

$$\{[a, b], [a, d], [b, d], [c, d]\}$$
[2]

and is indeed a subset of the set of pairs [1] of the total order. The subset property expresses that whenever the given constraints specify a certain order between two elements, the output of the algorithm must list these elements in that order.

This yields the definition of “topological sort”, stating simply that we must find a total order of which the given order relation is a subset. ← Page 413.

Acyclic relations have a topological sort

The absence of cycles is clearly a necessary condition for the existence of a topological sort (a total order that includes the original relation). What about the other way around: if we have an acyclic relation, can we always produce a topological sort — a total order that includes it?

The answer is yes:

Topological Sort theorem

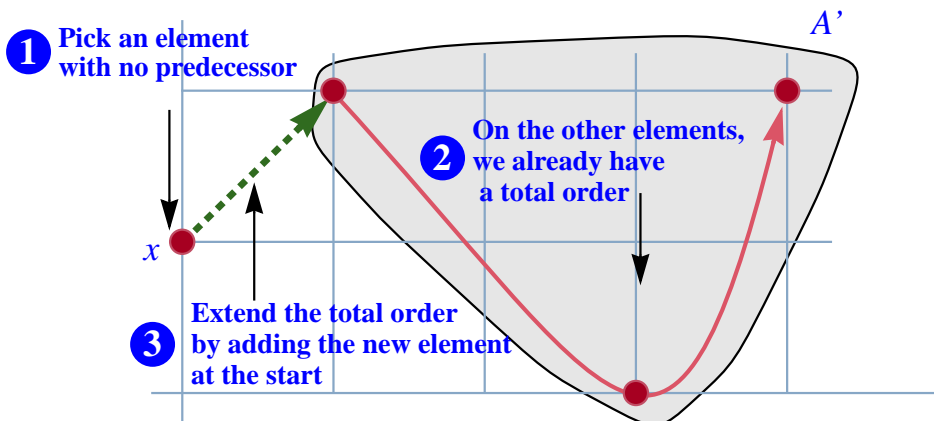
For any acyclic relation r over a finite set A , there exists a total order relation t over A such that $r \subseteq t$.

To prove this theorem we may use the observation that $r \subseteq r^+$, where r^+ is the transitive closure of r , and the previously proved property that r^+ is an order relation; it suffices to extend r^+ to a *total* order relation. But another proof is more interesting for our purposes. It is a *constructive* proof (relying on the No Predecessor theorem) and will allow us directly to deduce an algorithm scheme.

This proof is by induction on the number of elements n in the set A . If $n = 0$, the set is empty; the only possible relation is the empty relation (the empty set of pairs of elements of A), which is a total order. This proves the base step.

If you prefer, you can use as base step the case $n = 1$, for which A consists of a single element x ; even though A is not empty then, the only acyclic relation in A is the empty relation again, since if a relation has at least one pair that pair must be $[x, x]$, which would create a cycle.

For the induction step assume that the theorem holds for sets of n elements and consider an acyclic relation on a set A of $n + 1$ elements. The figure gives the idea of the proof:



Extending an incomplete topological sort

← “*Theorem: Acyclic and order relations (2)*”, page 417.

→ See exercise 17-E.5, page 452.

The No Predecessor theorem tells us that A has at least one element without predecessors. Let x be such an element. Let A' be the set consisting of all elements of A except x , and r' the relation on A' consisting of all pairs of r except those involving x . Clearly, r' is an acyclic relation over A' . By the induction hypothesis, since A' has n elements, there exists a total order t' over A' that is compatible with r' (that is to say, $r' \subseteq t'$). Now consider the relation t over A consisting of the following pairs:

- All the pairs in t' .
- All pairs of the form $[x, y]$ where y is an element of A' .

If you prefer to think of a total order as an enumeration, you may just view t as the enumeration of the elements of A that starts with x and continues with the enumeration of the elements of A' given by t' .

It is easy to see that t is a total order, and that $r \subseteq t$; this gives us a total order compatible with r , and proves the theorem.

The Topological Sort theorem is the mathematical justification for the program that we are now going to build; better yet, its proof directly suggests the algorithm's basic idea.

17.3 PRACTICAL CONSIDERATIONS

With the theoretical basis clear, we can start looking for a software solution. The core is a topological sort *algorithm*, but first we must examine performance constraints and define a software engineering framework.

Performance requirements

What can we expect to achieve in time and space complexity?

The inputs to the algorithm are a set of elements and a set of constraints. Let n be the number of elements and m the number of constraints.

The algorithm must (in the case of an acyclic relation) perform:

- At least one operation for every constraint (since ignoring any single constraint might make any particular output order wrong).
- At least one operation for every element, if only to add it to the output.

So the best time complexity that we may hope for is $\mathbf{O}(m + n)$.

What's more surprising is that the topological algorithm developed below actually achieves this theoretical ideal, both in time and in space.

Class framework

A purely algorithmic solution would use a function of the form

```
topologically_sorted (elements: ...; constraints : ...): LIST [...] is
  -- Enumeration of the members of elements,
  -- in an order compatible with the constraints
```

with appropriate input types to represent the sets *elements* and *constraints* (corresponding to our earlier *A* and *r*). It's better — as the remaining development will progressively show — to use an object-oriented approach with a class *TOPOLOGICAL_SORTER*, any instance of which represents an instance of the topological sort problem. The data structures representing the elements and constraints will be attributes of the class, set up through initialization procedures such as *record_element* and *record_constraint*. Instead of a function *topologically_sorted* as above we'll have:

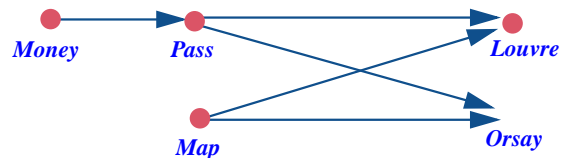
- A procedure *process* which performs the topological sort process.
- A query *sorted*, returning a list of elements, as computed by *process*.

This framework gives us more flexibility, and will accommodate many useful additional features.

Input and output

The sets of elements *A* and constraints *r* might come from many different sources. For example we might have a file listing the constraints, one per line:

```
Map    Louvre
Map    Orsay
Pass   Louvre
Pass   Orsay
Money  Pass
```



(From the figure on page 409.)

It may be useful to have a separate file listing all elements, or at least elements *not* involved in any constraint (we can't guess such elements from the constraints, but they should still be part of the output).

In another setup, the input might have been entered interactively using a program or a Web form. In examples such as ordering rectangles on a screen, terms in a glossary or features of a class, the format will be again different.

To preserve generality we make our basic class generic: it becomes *TOPOLOGICAL_SORTER* [*G*], where the parameter *G* represents the type of the elements. Then the result of the query *sorted* — denoting the topologically sorted list of elements computed by *process* — is of type *LIST* [*G*]; the two initialization procedures cited have signatures

```
record_element (e: G)
record_constraint (e, f: G)
```

Overall form of the algorithm

Consider an acyclic relation *r* over a set of elements *A*; since we need program names let's assume — without prejudging its implementation choices — that the class *TOPOLOGICAL_SORTER* has them available through queries *constraints* and *elements*. The general scheme for the topological sort algorithm in procedure *process* is:

```
from ... until elements.is_empty loop
  "Let x be an element without predecessors for the constraints"
  "Produce x as the next element of sorted"
  "Remove x from the set of elements"
  "Remove all pairs starting with x from the set of constraints"
end
```

Suitably refined, this form will work if we indeed start from an acyclic relation (or, as a special case, an order relation).

The No Predecessor and Topological Sort theorems give the justification, which we should express through a loop invariant and variant (did the spectacle of an invariant-less loop make you scream? — I hope it did):

```

process is
  -- Produce in sorted an enumeration of the members of elements,
  -- in an order compatible with constraints.
require
  -- “constraints describes an acyclic relation on elements”
do
  from
    create {...} sorted.make
  invariant
    -- constraints describes an acyclic relation on elements
  variant
    elements.count
  until
    elements.is_empty
  loop -- As before, except for explicit use of the result list sorted:
    “Let x be a member of elements without predecessors for constraints”
    sorted.extend(x)
    “Remove x from elements”
    “Remove from constraints all pairs starting with x”
  end
ensure
  -- “sorted is a topological sort of elements according to constraints”
end

```

Cycles in the constraints

This version of the algorithm scheme is correct in principle but not suitable for most real-life applications of topological sorting.

The problem is the precondition that assumes r to be an acyclic relation, that is to say, devoid of any cycles. A topological sort program gets its input in the form individual ordering constraints, for example [\[Map, Louvre\]](#), [\[Map, Orsay\]](#) as above. Such input may have been prepared by humans, and we can't be sure that it's error-free. (In industrial plant maintenance, there may be thousands of tasks and tens of thousands of constraints between them.)

In the glossary example, we may hope that no set of two or more terms all reference each other in their definitions (thereby creating a cycle), but we have no way to enforce this rule on glossary authors. The expectation is in fact the other way around: the glossary's author will expect a program that can be told: “*Order these entries so that definition always comes before first use — and by the way, if you find any mutually referential entries, tell me what they are, so that I can improve the definitions*”.

Similarly, the task of ordering the features of a class so that declarations appear before calls is impossible in the case of indirect recursion (direct recursion is fine), even though this is not an error. It may still be interesting to apply a topological sort algorithm to the non-cyclic part of the call graph, and report any remaining cycles.

These considerations suggest that *process* renounce its above contract

```
require
  -- “constraints describes an acyclic relation on elements”
ensure
  -- “sorted is a topological sort of elements according to constraints”
```

for a more realistic one:

```
-- (No precondition.)
ensure
  -- “sorted is a topological sort, according to constraints, of all the
  -- members of elements not involved in a cycle”
```

This is not enough yet, since the class should be able to report to its clients that the input contains a cycle — and what elements are involved. One way to provide this functionality would be through a boolean-valued function

```
has_cycle: BOOLEAN is
  -- Is the relation represented by elements and constraints
  -- not acyclic?
do ... end
```

or, more informatively, a function that returns the list of elements involved in a cycle (void if and only if *has_cycle* is false). This is conceptually sound, but not the best approach because it’s computationally too expensive. Finding cycles — the job of a function *has_cycle* — is essentially as hard, in time and space complexity, as topological sort proper; but if we *attempt* to do a topological sort without the precondition, we may at little extra cost find the cycles (the elements that violate the precondition).

The No Predecessor theorem tells us indeed how we can find cycles as a side bonus of a topological sort process:

- As shown in the loop above, we look at each stage, as long as the set of elements is not empty, for an element without predecessors.
- The theorem indicates that if the relation is acyclic we'll always find such an element.
- If we *cannot* find an element that has no predecessors and the set *elements* is not empty, we know — from the theorem — that the remaining elements are all involved in at least one cycle. We can terminate the algorithm and report that a full topological sort is impossible. This is a graceful form of termination, since we will have topologically sorted the elements that are not in cycles, and will be able (from the remaining *elements* and *constraints*) to report to the client which elements and constraints cause the problem.

In this scheme, used in the rest of this chapter, the topological sort routine has no precondition and the loop invariant, instead of

```
-- "constraints describes an acyclic relation on elements"
```

gets simplified to:

```
-- constraints describes a subset of the original relation on elements
```

with the consequences that

```
-- Any cycle in constraints was present in the original relation
```

and

```
-- constraints describes an acyclic relation if the original was acyclic
```

This is the basis we should retain. As a consequence, we can't any more use the loop exit condition *elements.is_empty* as above, since a non-empty *elements* no longer guarantees that we may correctly execute the instruction

```
"Let x be a member of elements without predecessors for constraints"
```

As new exit condition, we'll simply have

```
"No member of elements is without predecessors for constraints"
```

whose negation — there is at least an element without predecessors — guarantees that the loop body can find the next candidate element for output.

Overall class organization

We can now define the overall form of the class that will serve as the framework for the solution:

```

class
  TOPOLOGICAL_SORTER [G → HASHABLE]
feature {NONE} -- Internal data structures
  ... See next sections ...
feature -- Initialization
  record_element (a: G) is
    -- Include a in the set of elements.
  require
    not_sorted: not done
  do
    ... See next sections ...
  end
  record_constraint (a, b: G) is
    -- Include [a, b] in the constraints
  require
    not_sorted: not done
  do
    ... See next sections ...
  end
feature -- Status report
  done: BOOLEAN
    -- Has topological sort been performed?
feature -- Element change
  process is
    -- Perform a topological sort over all applicable elements.
    -- Results accessible through sorted, cycle_found and cyclists.
  require
    not_sorted: not done
  do
    ... See next sections...
  ensure
    sorted: done
  end

```

→ The routine body appears on page [430](#) (revised, page [440](#)).

```

feature -- Access
  cycle_found: BOOLEAN
    -- Did the original constraint imply a cycle?
  cyclists: LIST [G]
    -- Elements involved in any cycle
  sorted: LIST [G]
    -- List, in an order respecting the constraints, of all
    -- the elements that can ordered in that way

feature -- Status setting
  reset is
    -- Allow further updates of the elements and constraints.
  do
    done := False
    cycle_found := False ; cyclists := Void ; processed_count := 0
  ensure
    fresh: not done
  end

invariant
  elements_exist: elements /= Void
  constraints_exist: constraints /= Void
  cyclists_only_if_cycle: done implies (cycle_found = (cyclists /= Void))

end

```

The feature clauses have been listed in an order facilitating sequential reading rather than the recommended standard order, which a final version should respect.

The class is generic; the generic parameter *G* represents the type of elements. At this stage of the discussion, the elements can be of an arbitrary type; the reason for constraining *G* by *HASHABLE* will emerge as we devise the proper data structures.

→ “*Numbering the elements*”, page 435.

The algorithm will rely on internal data structures, which we’ll devise in the next sections; the corresponding features do not need to be available to clients, so they will all be declared under **feature** {*NONE*}.

Once *process* has done its job, it will make its results available to clients through several related queries:

- The boolean *done* — false after initialization — enabling clients to find out whether a topological sort has indeed been performed.
- The list *sorted*, giving an order compatible with the constraints for those elements that don’t participate in a cycle.

- The boolean *cycle_found*, to indicate whether any elements were determined to participate in one or more cycles.
- The list of all these cycle-involved elements, which we accordingly call *cyclists*. The invariant clause *cyclists_only_if_cycle* tells us that it's meaningful only if *cycle_found* is true.

So a typical use of the class by a client wishing to perform a topological sort is:

```

your_structure: TOPOLOGICAL_SORTER [YOUR_ELEMENT_TYPE]
...
create your_structure
... Calls of the form your_structure.record_element (x) to record elements
... and your_structure.record_constraint (x, y) to record constraints ...
your_structure.process
-- The topologically sorted elements are now available,
-- in the correct order, as your_structure.sorted.
if your_structure.cycle_found then
  -- The elements involved in cycles are now available
  -- through your_structure.cyclists ...
end

```

It would be desirable for consistency to equip the queries *sorted*, *cycle_found* and *cyclists* with the precondition *done*, but we omit it for the moment.

There is, however, a precondition **not** *done* for the initialization procedures *record_element* and *record_constraint*, as well as for the topological sort procedure *process* which has the postcondition *done*. This enforces the rule that as a client you should first set up the elements and constraints, then call *process*. As a result, it's an error to call *process* several times in succession on the same class instance; since the constraints won't have changed, this would make no sense (although you may of course reuse the query results as many times as you wish). The procedure *reset* is there in case you explicitly want to add elements and constraints after a call to *process*, in preparation for a new call to *process*.

Procedure *reset* simply sets *done* to false, without clearing the previous elements and constraints. We might add a procedure *forget* that calls *reset* and clears all data structures. But it's just as reasonable to assume that, in this case, the client will create a new instance of *TOPOLOGICAL_SORTER*.

17.4 BASIC ALGORITHM

We can now start to provide a full implementation of the key part of the solution, procedure *process*.

The loop

We already had a general algorithm for *process*; adapted in light of all subsequent observations (loosening the invariant, using the feature *sorted* which represents the result in *TOPOLOGICAL_SORTER*), it reduces to this: ← Page 424.

```

from
  create { ... } sorted.make
until
  “No element is without predecessors”
loop
  “Let x be an element without predecessors”
  sorted.extend (x)
  “Remove x from the set of elements”
  “Remove all constraints starting with x”
end

if “Any elements remain” then  -- Report cycle:
  cycle_found := True
  “Insert these elements into cyclist”
end

```

All that remains — don’t rejoice too soon, major decisions still lie ahead — is to refine the pseudocode elements into actual program text. The final part (reporting cycles) will be a straightforward consequence of the rest; this leaves the four highlighted operations, in fact just three since we can treat the first two (finding out if there’s an element without predecessors, and if so get one such element) as a single operation. They will be the focus of our search for a good algorithm:

Topological sort: the basic operations

- T1 • Find an element without predecessors — or report there isn’t any.
- T2 • Given an element *x*, remove it from the set of elements.
- T3 • Given an element *x*, remove from the set of constraints all that start with it (that is to say, all pairs of the form [*x*, *y*] for some *y*).

We must find a representation for the elements and constraints that makes these operations as efficient as possible. The data structures will go in as secret features in the section reserved for that purpose in the class text.

← The **feature** {NONE} clause on page 427.

That class sketch left two other routine bodies unfilled: *add_element* and *add_constraint*. We'll need to complete them based on the data structures that we'll have devised.

A “natural” choice of data structures

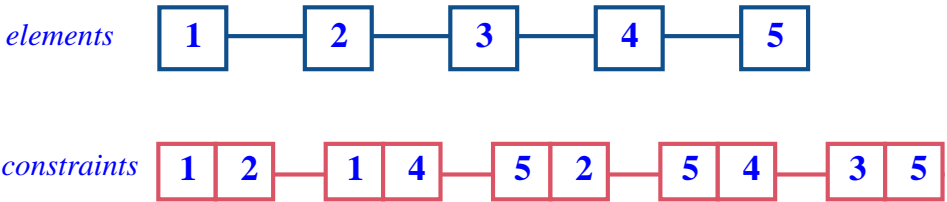
For our first attempt at data structures it is natural to choose a representation that directly models the problem's input as it comes to us. (One thing you may have already learned about programming is to become suspicious when you hear a solution presented as *natural*. What's natural to me may not be natural to you; and what's natural to you and me may turn out to be silly.)

We most likely get our data as a list of elements and a list of constraints. We can use attributes that directly reflect that structure (declared secret, as all data structures that follow, by appearing, in the class text, in a section of the the form **feature** {NONE} -- Internal data structures):

←Page 427.

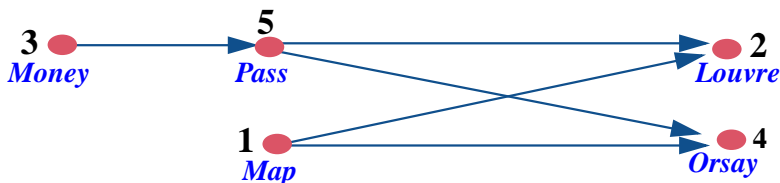
```
elements: LINKED_LIST [G]
constraints: LINKED_LIST [TUPLE [G, G]]
```

In our example the data structures will look like this:



Elements and constraints

assuming for convenience that we have assigned numbers to the example's elements, as follows:



Numbering the elements

(From the figure on page 409.)

Performance analysis of the natural solution

Can we implement what we need — the operations [T1](#), [T2](#), [T3](#) and the procedures *record_element* and *record_constraint* — with this representation, and if so what is the time and space cost?

The two procedures are straightforward. For example *record_constraint* (x, y) will just perform

```
constraints.extend ([ $x, y$ ])
```

adding the tuple [x, y] at the end of the list of constraints. Similarly, *record_element* (x) will perform *elements.extend* (x).

We must make sure that *constraints* and *elements* are non-void for such instructions; the corresponding *create* instructions may either appear in a *default_create* for the class, or be performed on demand on first need. This will also apply to other similar data structures introduced below.

We can also use this representation to perform the other operations; let's examine the cost:

- To find if there's an element without predecessors ([T1](#)), we can traverse the list of *constraints* and count predecessors, then traverse the list of elements to find those for which the count is zero; but the first part is an $\mathbf{O}(m)$ operation and the second $\mathbf{O}(n)$. If we do this at every step, the total cost is $\mathbf{O}(m * n + n^2)$.
- Removing an element ([T2](#)) can be as bad as $\mathbf{O}(n)$ each time (meaning $\mathbf{O}(n^2)$ for the whole process) with a linked list, although we could bring it down to $\mathbf{O}(1)$ ($\mathbf{O}(n)$ total) through simple data structure adaptation.
- Removing a set of constraints ([T3](#)) can again be as bad as $\mathbf{O}(m)$ each time, meaning $\mathbf{O}(m * n)$ altogether, if all we have to represent constraints is the global list *constraints*, which doesn't enable us to find all the constraints starting with a given x without traversing the whole structure.

Anything that's $\mathbf{O}(m * n)$ or $\mathbf{O}(n^2)$ is really bad. In particular, we may in a practical application expect most elements to be involved in at least one constraint — often many more in the average, e.g. ten or so in a typical scheduling problem —, so that $m > n$, implying that $\mathbf{O}(m * n)$ is worse than $\mathbf{O}(n^2)$. Anything that's in n^2 , growing like the square of the number of elements, will be out of reach for large practical applications, as the number of elements may be quite large.

So with this first choice of data structures we do have a solution, but performance-wise it doesn't scale up to large practical problems.

Duplicating the information

Fortunately we can do better than the “natural” solution. The observation is that we don’t have to be lazy and use the data structures as they’re given to us. The lists *elements* and *constraints*, express the data in a form that directly mirrors how things look to the external world, for example to the person who inputs a set of tasks and a set of associated constraints. What’s clear and “natural” to describe the input to the outside world is not necessarily the best form for an *algorithm* that will process the data for a specific purpose. Rather than following the original form blindly, the algorithm may start with an initialization phase that turns it into the format best suited to that processing.

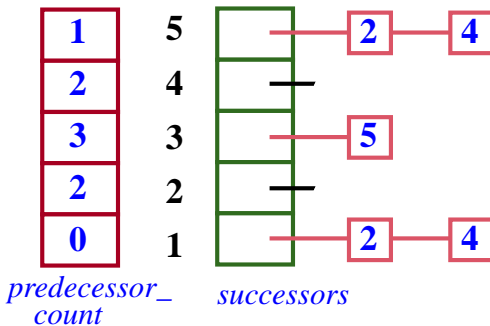
The following data structures help make the job of topological sort — tasks **T1** to **T3** — convenient and fast:

```

successors: ARRAY [LINKED_LIST [INTEGER]]
    -- Indexed by element numbers; for each element x, gives the list of
    -- its successors: the elements y such that there is a constraint [x, y].

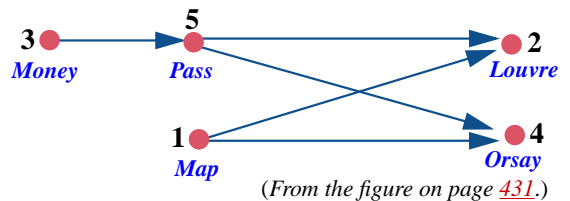
predecessor_count: ARRAY [INTEGER]
    -- Indexed by element numbers; for each, says how many
    -- predecessors the element has.
    
```

Here’s how they will initially look for our working example:



Number of predecessors, and lists of successors

This expresses the constraints between elements, repeated on the side figure. For example the explanation for the entries of index 1 and 2 is that task 1 (*Map*) has no predecessor and the successors 2 and 4 (*Louvre* and *Orsay*), and that element 2 (*Louvre*) has two predecessors and no successors.



What's interesting in this representation is that the array *predecessor_count* is conceptually redundant: we could always reconstruct the information it provides by exploring the array *successors*, which includes all there is to know about the constraints. But there's nothing wrong with storing information in two (or more) different ways if — as we're going to see — it brings us a significant improvement in computation time.

Such **space-time tradeoffs** are a key ingredient of good algorithm design. Of course the tradeoff has to be acceptable. Here our goal is to have $\mathcal{O}(m+n)$ time complexity. In space complexity, *successors* is $\mathcal{O}(m+n)$ (one array entry per element, one tuple and reference per constraint); adding the $\mathcal{O}(n)$ array *predecessor_count* doesn't change the picture.

The original data structures, *elements* and *constraints*, already took up $\mathcal{O}(m+n)$ space.

Spicing up the class invariant

It is convenient for clarity to add a query

```
count: INTEGER
    -- Number of elements
```

which we can make public. It is also useful, if only for readability, to add the following invariant clauses, the last two expressed informally:

```
elements.count = count
predecessor_count.count = count
successors.count = count

-- For every i in  $1..count$ : predecessor_count [i] is the number of
-- predecessors of i according to the constraints.

-- For every i in  $1..count$ : successors [i] contains all the successors
-- of i as implied by the constraints, or is void if i has no such successors.
```

Numbering the elements

To use an array we need to associate an integer with every element. I sneakily introduced this convention a while ago but now it's not just a useful convention; it's required by our choice of data structures.

← “A “natural” choice of data structures”, page 431.

Does this mean that we should renounce the generic parameter G of our class `TOPOLOGICAL_SORTER [G]` since all manipulations of elements will now use their integer numbers? Absolutely not. It's still necessary, for expressiveness, to have a mechanism applicable to elements of any type. All we need in practice is a hash table and an array

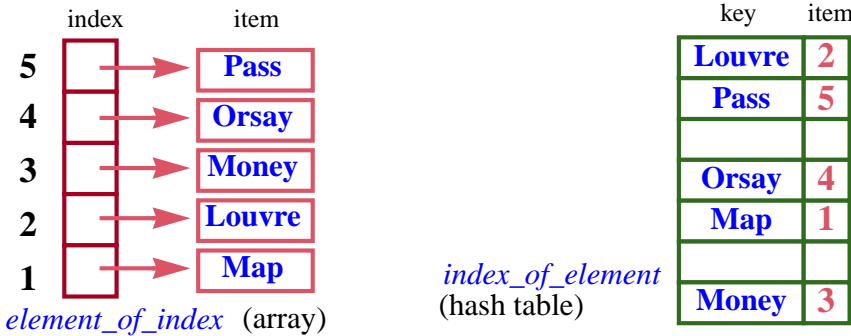
```

index_of_element: HASH_TABLE [INTEGER, G]
    -- For every element, gives its index
element_of_index: ARRAY [G]
    -- For every assigned index, gives the associated element
    
```

The integer `index_of_element [e]` will be the number x assigned to an element e , of type G . Then `element_of_index [x]` will be e .

← “Bracket notation and assigner commands”, page 261.

Both of these use bracket notation: `index_of_element [e]` is the item of key e in the hash table, and `element_of_index [i]` is the item of index i in the array.



Two-way mapping between elements and numbers

Subject to a proper implementation of hash tables, these structures are both $O(n)$ in space.

To define a hash table of elements of type G requires that G conform (through inheritance) to class `HASHABLE`. This was taken care of by declaring the class as `TOPOLOGICAL_SORTER [G -> HASHABLE]`.

← Page 427.

`TOPOLOGICAL_SORTER` will not export `index_of_element` and `element_of_index`, since these features are for implementation only; but we must enable clients to find out if a certain element is part of the problem, and hence export the following query:

```

has_element (e: G): BOOLEAN is
  -- Is e one of the elements to be topologically sorted?
  do
    Result := index_of_element.has (e)
  ensure
    consistent: Result = (index_of_element.has (e) and then
      index_of_element [e] >= 1 and then
      index_of_element [e] <= element_of_index.count and then
      element_of_index [index_of_element [e]] = e)
  end

```

It's worthwhile to make sure you understand the postcondition.

Let's see how our new data structures help reach the goal of $\mathbf{O}(m+n)$ time. Two aspects are now involved: operations **T1** to **T3**, but also initializing the data structures *predecessor_count* and *successors*. Both are equally relevant: it wouldn't help to have $\mathbf{O}(m+n)$ for the core of the algorithm (the loop iterating operations **T1** to **T3**) if the initialization took — say — $\mathbf{O}(m * n)$.

It doesn't seem too hard to initialize the data structures in $\mathbf{O}(m+n)$: process all constraints in sequence; for every constraint $[x, y]$, increment the y entry in the array *predecessor_count*, and insert y into the list *successors* $[x]$. Both of these operations are $\mathbf{O}(1)$, so applying them to all constraints is $\mathbf{O}(m+n)$. We'll need to spell out the details, but for now let's indeed assume $\mathbf{O}(m+n)$ initialization and concentrate on **T1** to **T3**, the core operations of the algorithm.

→ Exercise [17-E.6](#), page 452.

→ “*Initializations and their time performance*”, page 442.

Basic operations

Let's start with **T3**: “given an element x , remove all constraints of the form $[x, y]$ for any y ”. If we know the number for x , this is straightforward:

- L1 • We won't need the list of successors of x any more. We could make it void through *successors* $[x] := \text{Void}$. In practice, this is not necessary, as the algorithm will simply never visit the entry x of *successors* any more. But even if we had to perform this operation it would be $\mathbf{O}(1)$ — meaning $\mathbf{O}(n)$ globally if we apply it to all elements. Good!
- L2 • We must also update any relevant entry in the array *predecessor_count*. The effect on this array of “removing all constraints $[x, y]$ ” for given x means that we must decrease *predecessor_count* $[y]$ by 1 for every successor y of x . So it suffices to traverse the list *successors* $[i]$ (before you set it to void, of course, if you do want to do that), and for each element encountered decrease the corresponding entry in *predecessor_count*. This is a straightforward loop, whose code appears below. This process will be done *at most once*, in the entire processing, for each constraint in the system. So it is $\mathbf{O}(m)$. Good again!

→ Innermost loop on page 440.

T3, then, is $\mathbf{O}(m + n)$ at worst.

All the processing just described is there to maintain the invariant clauses expressing that the array *predecessor_count* and the array of lists *successors* faithfully reflect the structure of the remaining constraint relation.

← As introduced on page 434.

T2 is “given an element x , remove it from the set of elements”. In fact, with our new data structures, we don’t really need to do anything here. The information that really matters affects the *constraints* starting with x , and we’ve just taken care of these. Definitely good.

There remains **T1**, “find an element without predecessors — or report there isn’t any”. It suffices to traverse the array *predecessor_count* and look for zero values. But this is $\mathbf{O}(n)$ meaning, overall, $\mathbf{O}(n^2)$. Not good!

We are still missing one — our last — data structure.

The candidates

We won’t avoid one $\mathbf{O}(n)$ traversal of the array *predecessor_count* upon initialization — the **from** clause of our main loop — to find out the initial “candidates” for immediate output: elements without predecessors in the original relation. Unless *every* element is involved in some cycle, a rather inauspicious initial situation for an attempt at even partial topological sorting, we’ll find one or more x for which *predecessor_count* [x] is initially 0. This requires $\mathbf{O}(n)$, as noted, but paying $\mathbf{O}(n)$ once is not a problem.

→ In procedure *find_initial_candidates*, page 441.

After that we won’t ever need to traverse the array *predecessor_count* to look for new candidates. It suffices to notice that the operation labeled **L2** above, which decrements one or more entries of the array *predecessor_count*, is the *only* one that can make an entry of the array zero if it wasn’t zero initially. So we’ll just extend the operation so that it watches for entries that become zero. Assuming it was written

```
-- Decrease the y entry of predecessor_count by one:
predecessor_count [y] := predecessor_count [y] - 1
```

we replace it by

```
-- Decrease the y entry of predecessor_count by one
-- and check if this makes y an element without predecessors:
predecessor_count [y] := predecessor_count [y] - 1
if predecessor_count [y] = 0 then
    “Record that y is now without predecessors”
end [3]
```

To “Record that an element is without predecessors” it suffices to add it to a structure *candidates* which will, after initialization and each iteration of the loop, contain all elements, not yet processed, that have no predecessor. What concrete data structure should we use for *candidates*? For the topological sort algorithm the precise choice doesn’t matter as long as the structure supports the following five features:

count is there for completeness, but we won’t actually need it.

```

feature -- Access
    item: G
        -- An element previously inserted
    require
        not_empty: not is_empty

feature -- Measurement
    count: INTEGER
        -- Number of elements
    ensure
        non_negative: Result >= 0

feature -- Status report
    is_empty: BOOLEAN
        -- Is there no element?
    ensure
        definition: Result = (count = 0)

feature -- Element change
    put (x: G)
        -- Insert x.
    ensure
        one_more: count = old count + 1

    remove: G
        -- Remove the element given by item.
    require
        not_empty: not is_empty
    ensure
        one_fewer: count = old count - 1

```

The general name for such a structure is **dispenser**, by analogy with a machine into which you may deposit elements (*put*) and also, by pressing a button, getting a previously deposited element (*item* and *remove*), assuming there is still at least one (**not** *is_empty*):

A dispenser

(Insert picture of piggybank)

Unlike with an array or list, you don't choose the element to get and remove: the dispenser chooses for you. A stack is a dispenser, with a LIFO behavior (you get the element most recently deposited); a queue is also a dispenser, with a FIFO behavior (you get the oldest not yet removed element). There are many other possible policies.

For topological sort, any dispenser will do the job. Choosing a particular kind affects the actual order — among those compatible with the constraining relation — in which the algorithm outputs elements. This is the key lever that you can apply to select a specific policy, for example to ensure that the result will optimize a certain criterion. It's also the reason for describing topological sorting as an algorithm *family* rather than a single algorithm.

→ As explored in exercise [“Parameterizing topological sort”](#), 17-E.9, page 452

We'll declare the candidate dispenser as

```
candidates: PRIORITY_QUEUE [INTEGER]
    -- Elements without predecessors, ready to be released

-- Additional clause for the invariant:
    -- For every item x of candidates, predecessor_count [x] = 0
```

An implementation of *STACK* or *QUEUE* would also do; a “priority queue” is a more general kind of dispenser where every element may be given a priority, with the rule that *item* yields (and *remove* takes away) the element with the highest priority. *STACK* and *QUEUE* are the special cases of priorities set as an increasing and decreasing function of the order of insertions. A general *PRIORITY_QUEUE* allows you, by playing with the priorities, to set the selection policy that you wish.

The loop, final form

We may now write the **main loop** of the topological sort algorithm — the body of the procedure *process* — with all its details. The pseudocode instructions of the previous version have been left as comments (in red) for comparison. The routine must declare local variables *x* and *y* of type *INTEGER* and *x_successors* of type *LIST [INTEGER]*, recording the successors of a particular element. We also add an integer variable *processed_count* — used next — to keep track of how many elements we have processed.

← The basic form appeared on page 430. For the context, including procedure *process*, see page 427.

```

from
  create sorted.make
  find_initial_candidates      -- See next
invariant
  -- “The data structures represent a subset of the original elements,
  -- and the corresponding subset of the original relation”
until
  candidates.is_empty
loop
  -- “Let x be a member of elements with no
  -- predecessor for constraints”
  x := candidates.item ; candidates.remove
  sorted.extend (element_of_index [x])
  -- “Remove x from elements and
  -- all pairs starting with x from constraints”
  x_successors := successors [x]      -- A list
  from x_successors.start until x_successors.after loop
    y := x_successors.item
    -- Next few lines are from [3], page 437:
    predecessor_count [y] := predecessor_count [y] - 1
    if predecessor_count [y] = 0 then
      -- “Record that y is now without predecessors”
      candidates.put (y)
    end
    x_successors.forth
  end
  processed_count := processed_count + 1
variant
  count - processed_count
end
report_cycles      -- See next
done := True

```

This algorithm assumes that the arrays *predecessor_count* and *successors* have been properly set up, as must be the case before any call to *process*. The details of the initializations are coming next.

→ “Initializations and their time performance”, page 442 below.

Operations on *candidates* are highlighted to emphasize how critical this structure has now become to the algorithm.

The procedure *find_initial_candidates* must set up the *candidates* dispenser with the elements initially without predecessors. It’s straightforward:

```

find_initial_candidates is
  -- Insert into candidates any elements without predecessors.
  local
    x: INTEGER
  do
    if candidates = Void then create candidates end
    from x := 1 until x > count loop
      if predecessor_count [x] = 0 then
        candidates.put (x)
      end
      x := x + 1
    end
  end
end

```

This is the $O(n)$ traversal that without *candidates* we would have had to perform at each step of the loop; now we just do it once at the beginning. It’s not an error for the procedure to find *no* elements satisfying *predecessor_count* [x] = 0, but will simply result in an empty *candidates* structure, causing the loop to terminate immediately, as every element is involved in a cycle.

← As noted at the end of “Basic operations”, page 437.

Procedure *process* must do one more thing after the loop: set up the information enabling a client to find out about any cycles in the input. This is the task of procedure *report_cycles*. To implement it, we note that the loop terminates when there are no more elements in *candidates*; if the original relation was acyclic we will have processed all elements, so we use *processed_count* to find out whether there’s any left:

```

report_cycles is
  -- Make information about cycles available to clients.
do
  if processed_count < count then
    -- There was a cycle in the original relation!
    cycle_found := True
    create {LINKED_LIST [G]} cyclists.make
    from x := 1 until x > count loop
      if predecessor_count [x] /= 0 then
        -- x was involved in a cycle
        cyclists.extend (element_of_index [x])
        x := x + 1
      end
    end
  end
end

```

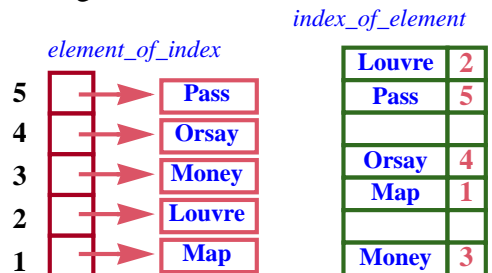
Initializations and their time performance

We now have an efficient — $O(m + n)$ — implementation of the core of the topological sort loop, thanks to three data structures chosen directly to fit its needs: the arrays *predecessor_count* and *successors*, and the dispenser *candidates*. To complete the job we must spell out their initialization, making sure we don't exceed the performance constraints.

The initialization will have to perform:

- *record_element* (*e*) for every element: *n* times altogether.
- *record_constraint* (*e*, *f*) for every constraint: *m* times altogether.

The job of *record_element* (*e*) is to assign a number to *e*, so that the rest of the processing can deal with integers, rather than actual elements of *G*. This is done by filling in twin entries in the array *element_of_index* and the hash table *index_of_element*:



(From the figure on page 435.)

```

record_element (e: G) is
    -- Add e to the set of elements, unless already present.
require
    not_sorted: not done
do
    if not has_element (e) then
        count := count + 1

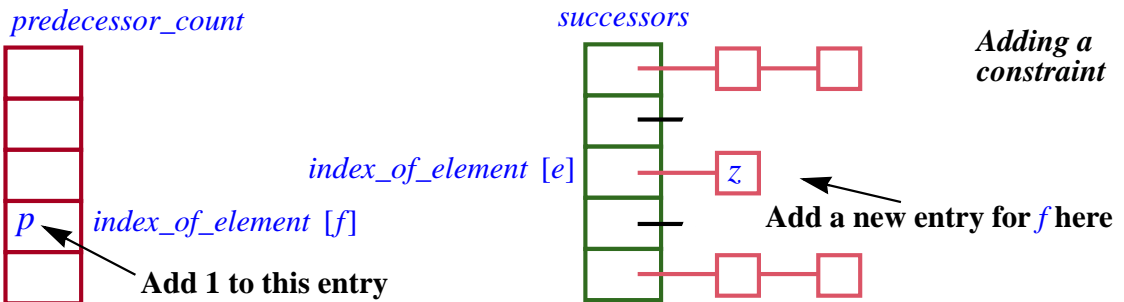
        index_of_element.extend (count, e)
        element_of_index.force (e, count)
        -- extend and force expand the structures if necessary; this means
        -- we don't need to know the number of elements in advance.
    end
ensure
    inserted: has_element (e)
    one_more: not (old has_element (e)) implies (count = old count + 1)
end
    
```

The initial test ensures that the procedure ignores a second attempt to insert a given element. This policy allows *record_constraint* (*e*, *f*) to start by calling *record_element* on both *e* and *f* just to make sure the elements are properly inserted. An exercise asks you for a way to avoid the duplication of work between *has_element* and *extend*.

→ Exercise 17-E.7, page 452.

With appropriate implementations of *extend* and *force*, the code of *record_element* is $O(1)$; executed for all elements, it will contribute $O(n)$ to the algorithm. This is in line with our requirements.

The remaining initialization mechanism is the procedure for entering constraints. A call to *record_constraint* (*e*, *f*) must increase by 1 the number of predecessors of *f* in the array *predecessor_count*, and add *f* to the list of successors of *e*. That list is one of the entries of the array *successors*:



The routine will read:

```

record_constraint (e, f: G) is
    -- Add the constraint [e, f].
require
    not_sorted: not done
    exist: e /= Void and f /= Void
local
    x, y: INTEGER
do
    -- Ensure e and f are inserted (no effect if they already were):
    record_element (e); record_element (f)

    x := index_of_element [e]
    y := index_of_element [f]
    predecessor_count [y] := predecessor_count [y] + 1
    add_successor (x, y)
ensure
    both_there: has_element (e) and has_element (f)
end

```

with an auxiliary procedure (which doesn't need to be exported):

```

add_successor (x, y: INTEGER) is
    -- Record y as successor of x.
require
    1 <= x ; x <= count
    1 <= y ; y <= count
local
    x_successors: LINKED_LIST [INTEGER]
do
    x_successors := successors [x]

    -- The successor list for x may not have been created yet:
    if x_successors = Void then
        create x_successors.make
        successors [x] := x_successors
    end

    x_successors.extend (y)
end

```

As suggested earlier, *record_constraints* starts by calling *record_element* on the constraint's two elements; because of the way we've designed *record_element*, this has no effect if they were already there. This policy makes it possible for a client application to start from just a list of constraints, never having to call *record_elements* explicitly.

We cannot, however, assume this will always be the case and remove *record_element* from the public interface of the class. An instance of the problem may, as noted, include elements that are not involved in any constraint but should still be listed as part of the output. In such a setup, the input must include, separate from the list of constraints, a list of elements.

On the subject of duplication, the procedure *record_constraint* does not attempt to determine if a constraint has already been entered. Indeed, as you are invited to check, our topological sort algorithm will work as expected if a constraint appears twice. This may well happen with manually entered data and the algorithm doesn't consider it an error. (There's nothing contradictory in saying twice "*e* must come before *f*".) To apply a different policy is the responsibility of the client application, as part of input validation.

Now what about efficiency? The code for each of the two auxiliary procedures is $O(1)$: one array access plus, in the second case, insertion at the end of a list (with a good implementation ensuring that the list cursor stays at the end) and, once for each applicable element, an object creation. As a result *record_constraint* as a whole is $O(1)$; as it's executed once for each constraint, its contribution to the algorithm is $O(m)$. We have achieved our goal of $O(m+n)$ time for the initialization as well as the main part of the algorithm.

Putting everything together

You have now seen all the program elements needed to implement topological sort. A class built directly from this discussion is available in EiffelBase and used in Traffic, but I suggest that independently of this existing implementation you check your understanding of the concepts by writing a class that brings them all together:

Programming time! **Usable implementation of topological sort**

From the elements of this chapter, write a class *TOPOLOGICAL_SORTER* providing a general, usable implementation of topological sort.

→ Exercises [17-E.8](#),
[page 452](#) and [17-E.9](#),
[page 452](#).

Make sure to engineer the solution properly by providing not just the algorithm but also the initialization procedures (*record_element*, *record_constraint*).

To test your solution, you will find at the URL for this course a file listing a few hundred example constraints, and all its possible topological sorts.

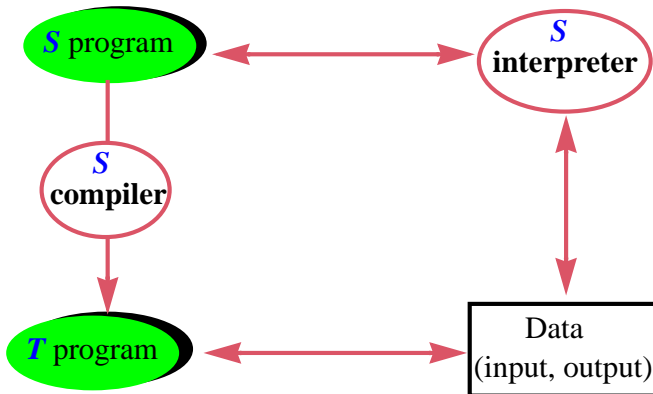
17.5 LESSONS

The topological sort algorithm has important consequences to teach us for both algorithm design and software engineering.

Interpretation vs compilation

In processing programs — written in some programming language S — for execution on a computer, two general styles of solution exist:

- Interpretation: write a program, called an **interpreter**, that can directly execute an arbitrary S program applied to an arbitrary input.
- Compilation: write a program, called a **compiler**, that transforms any S program into a program with equivalent semantics expressed in a target language T . If T is machine language for the desired platform, the result can be directly executed.



*Interpretation
and
compilation of
programs*

The two styles are often combined in practical language implementations. In particular the target T of a compiler doesn't have to be machine language, although it should be *closer* to machine language than S . Then T programs can be executed through direct interpretation, or through further compilation.

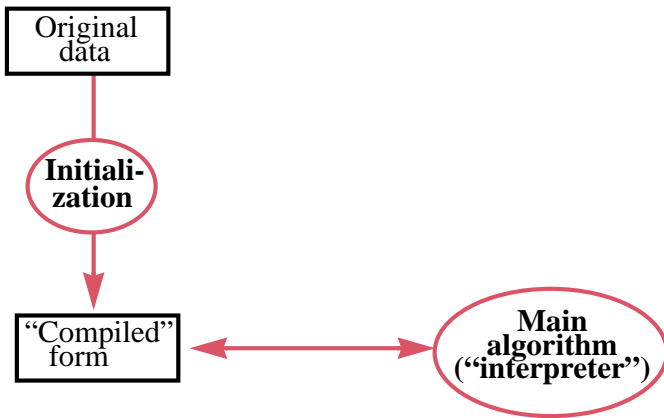
These concepts generalize to many application domains other than programming language processing. To perform a certain processing on certain input, we may use data structures that directly mirror the input; or we may proceed in two steps:

- *Compilation*: transform the data into a form more suitable for the algorithm's needs.
- *Interpretation*: apply the needed operations to the resulting structure.

This technique (which, as for language processing, may in the general case involve several iterations of the process) is exactly what we have applied for topological sort. We first looked at an “interpretive” solution using the seemingly natural data structures, directly deduced from the statement of the problem; but they turned out to yield bad performance for topological sorting. “Compiling” them into a representation directly tailored to our goals led to a solution with excellent performance.

← “A “natural” choice of data structures”, [page 431](#).

In such a two-step solution, the “compilation” step, which initializes the data structures, may be as delicate to devise as the actual processing based on its results, and it may account for as much time, sometimes more. That’s OK as long as the overall performance meets your goals — but of course you must not jump to conclusions about performance until you have taken into consideration the initialization as well as the later processing.



*“Interpretation”
and “compilation”
of data*

The approach can be summed up as a *heuristics* — a general strategy, similar to a “design pattern” but of a more abstract nature, that is known to help devise good solutions in suitable cases:

Touch of Heuristics: Compile the data first!

Good algorithms are often obtained through a two-step strategy where:

- The first step turns the input, from its given form, into an internal data structures carefully devised to suit the algorithms’ goals.
- The second step processes the resulting form to attain these goals.

Time-space tradeoffs

Closely tied to the “Compile the data first!” heuristics is the observation that the ideal data structure — the one best suited to the needs of the second step — is often not the most economical representation possible for the underlying information. In the example, information about constraints may end up being reflected in *three* different parts of the data structure: a constraint $[x, y]$ causes y to appear in the list of x 's successors in the array *successors*; it adds one to *predecessor_count* $[y]$; and the absence of any such constraint for a given y leads to inserting y into *candidates*. Such replication sacrifices some space to ensure a considerable gain in execution time. Tradeoffs of this kind, sometimes going the other way, are one the keys to efficient algorithm design.

← For the terminology see [“Information and data”](#), page 10.

Algorithms vs systems and components

It is possible to give a description of topological sort that ignores many of the aspects studied in this chapter, concentrating only on the final algorithm. For a usable solution, however, one must take into account the practical needs of applications. The object-oriented approach plays a key role in enabling us to meet this goal: instead of writing a “topological sort program” we have devised a **class**, *TOPOLOGICAL_SORTER*. An instance of this class describes an instance of the topological sort problem, equipped with not only the algorithm (procedure *process*) but also with all the apparatus allowing clients to:

- Set up the problem instance, by recording elements and constraints in a convenient way.
- Apply *process* to produce a topological sort of the applicable elements, satisfying the constraints.
- Query the resulting state, to discover whether any cycles were found, and if so what elements they involve.

The difference between this approach and a mere algorithm is part of the difference between **software engineering** and mere programming. In software engineering it is not enough to devise clever algorithmic solutions and the associated data structures; the goal is to provide *solutions* that can be integrated into successful systems.

The really desirable goal is to make these solutions **reusable**, so that they are not just “design patterns”, which programmers can integrate into their systems by buying and reading books (especially excellent books such as the present one), but *components* that can be made available, once and for all, for reuse directly off the shelf.

You will also have noted how, in this process, the **contracts** enable us at each step to know exactly what we are doing — what we expect, what we guarantee and what we maintain.

17.6 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A topological sort is an enumeration of a set of elements compatible with a set of ordering constraints on these elements.
- The problem has a simple mathematical description: given a (strict) order relation, find a total order that is a subset of it.
- In practice the given relation is usually not an order relation but just acyclic. Taking its transitive closure gives an order relation.
- A realistic, well-engineered software solution must accept possibly erroneous input in which the relation has cycles. It should then produce a topological sort of the acyclic part, and report remaining cycles.
- Such a solution must provide not just the topological sort algorithm but also mechanisms to build a problem instance by entering individual elements and constraints.
- With n elements and m constraints, it is possible to perform topological sorting in $O(m + n)$ time and space.
- The key to the efficiency of the algorithm is that it works from data structures specifically adapted to the problem: two arrays giving, for each element, the list of its successors and the number of its predecessors; and a dispenser (stack, list or priority queue) containing the set of elements without predecessors.
- As this example illustrates, good algorithmic solutions are often obtained by first “compiling” the problem’s data into a specially designed data structure, which can then be “interpreted” efficiently.

New vocabulary

Acyclic	Antisymmetric	Binary relation
Cycle	Irreflexive	Order relation
Partial order	Relation	Strict order
Topological sort	Total order	Transitive closure

17.7 TERMINOLOGY NOTE: ORDER RELATIONS

To discuss topological sort it is convenient — as this chapter has shown — to use *strict* order relations, as in “*strictly* less than”, such as the “ $<$ ” relation on numbers. For other problems it may be more useful to deal with the nonstrict versions, such as “ \leq ” on numbers. The two are closely related: $x \leq y$ holds if and only if $x < y$ or $x = y$. The common convention is for “order relation” to mean the nonstrict version. In this chapter, since we have used only strict order relations, the word “strict” has usually been omitted, so that “order” means “strict order”.

For a strict order relation (irreflexive and transitive), some of the literature uses the term *quasi-order*. Of course one may pick any name for a notion as long as one provides a precise definition, but this one is unfortunate since there is nothing “quasi” about such orders; if anything they are “more” ordered than nonstrict variants — those usually called “order relations” — since they don’t hold between an element and itself (irreflexivity). To make things worse, other authors use “quasi-order” for relations that are transitive and *reflexive* (rather than irreflexive). So it’s better to stay away from this term and instead qualify order relations as “strict” when needed.

The next issue is whether the relation is total or not. Totality means that for any two non-equal elements x and y one of $x < y$ and $y < x$ will hold. An order relation that satisfies this property is a *total order*. One that doesn’t satisfy the property — meaning that there’s at least one pair of distinct elements for which neither $[x, y]$ nor $[y, x]$ is in the relation — should be called a *partial order*. But that’s not what “partial order” relation means in most of the literature: it means an order relation that we don’t know to be total. In other words, it’s a *possibly partial* order relation. That’s confusing, since now a total order relation is also partial! It is better to write, as in this chapter:

- *Total order* for an order relation that is known to be total.
- *Partial order* for an order relation that is known to be non-total.
- If we don’t know, or want to include both cases, just *order*. In case of possible ambiguity, use “possibly partial order”.

17-E EXERCISES

17-E.1 Vocabulary

Give a precise definition of each of the terms in the vocabulary list on the preceding page.

17-E.2 Irreflexivity and asymmetric

Order relations were defined as irreflexive and transitive, and proved asymmetric as a consequence. Prove that it’s equivalent to define them as asymmetric and transitive, with irreflexivity a consequence. ← Page 415.

17-E.3 Total order and enumeration

Prove that if a relation r is a total strict order on a finite set, there exists a single enumeration of the elements such that, for any elements x and y , x appears before y in the enumeration if and only if the pair $[x, y]$ is in r .

17-E.4 Strict vs. nonstrict orders

The discussion in this chapter has relied on *strict* order relations (partial or total), such as " $<$ " on integers, "less than". It is also possible to use *nonstrict* order relations, such as " \leq ", "less than or equal to". The definition of a partial strict order — which we'll call " $<$ " although it doesn't have to be the usual relation on numbers — was that it must be:

← "[Binary relations](#)",
page 413.

O1 • **Irreflexive**: $x < x$ holds for no x .

O2 • **Transitive**: whenever $x < y$ and $y < z$ hold, so does $x < z$.

That the relation is also

O3 • **Asymmetric**: $x < y$ and $y < x$ may not both hold.

is a consequence of the previous two properties.

For a partial *nonstrict* order relation " \leq ", there are three independent conditions:

O5 • **Reflexive**: $x \leq x$ for any x .

O6 • **Transitive**: whenever $x \leq y$ and $y \leq z$ hold, so does $x \leq z$.

O7 • **Antisymmetric**: whenever $x \leq y$ and $y \leq x$ both hold, then $x = y$.

For any partial strict order relation " $<$ " there is an associated relation " \leq ", defined by

$x \leq y$ if and only if: $x < y$ or $x = y$	[4]
--	------------

Conversely, given a partial nonstrict order relation " \leq ", we may define an associated relation " $<$ " by

$x < y$ if and only if: $x \leq y$ and $x \neq y$	[5]
--	------------

This exercise explores the relationship between these associated strict " $<$ " and nonstrict " \leq " variants.

- 1 • Prove that if " $<$ " is a partial strict order relation, then " \leq ", as defined by [4], is a partial nonstrict order relation.
- 2 • Prove that if " \leq " is a partial nonstrict order relation, then " $<$ ", as defined by [5], is a partial strict order relation.

- 3 • In the strict case, the definition imposes only two conditions: irreflexivity and transitivity; condition [O3](#), asymmetry, is a consequence. In the nonstrict case, there are three conditions. Prove that antisymmetry, [O7](#) does not necessarily follow from the other two, reflexivity and transitivity. (In other words, find an example that satisfies [O5](#) and [O6](#) but not [O7](#))
- 4 • Prove that replacing “reflexive” by “irreflexive” in the definition of a strict order yields the definition of a nonstrict order.
- 5 • Prove that replacing “irreflexive” by “reflexive” in the definition of a nonstrict order yields the definition of a strict order.

17-E.5 Acyclic and total order relations

Prove the Topological Sort [theorem](#) on the basis of the [second theorem](#) on acyclic and order relations. ← [Page 420](#); [page 417](#).

17-E.6 An interesting postcondition

Explain the [postcondition](#) of the function [has_element](#). ← [Page 436](#).

17-E.7 Optimizing hash table usage

The [algorithm](#) for procedure [record_element](#) tests whether an element e is already present in the hash table [index_of_element](#), and inserts it only if not. This causes two search operations, one for [has](#) (called by [has_element](#)) and one for [extend](#). Examining the contract form of [HASH_TABLE](#) to find the appropriate features, rewrite the procedure to avoid this small inefficiency. ← [Page 444](#).

17-E.8 Programming topological sort

[Implement](#) the class [TOPOLOGICAL_SORTER](#) class according to the discussion of this chapter. → See [“Programming time! Usable implementation of topological sort”](#), [page 445](#).

17-E.9 Parameterizing topological sort

(This exercise assumes you have done the preceding one.) Extend the implementation of topological sort to enable clients to select a specific [policy](#) for choosing between competing “candidates” ready for output. → As discussed in [“The candidates”](#), [page 437](#).

PART IV:

Object-Oriented Techniques

We

18

Inheritance

18.1 ENFORCING A TYPE: OBJECT TEST

Assignment attempt

19

Operations as objects: agents and lambda calculus

The object-oriented framework has already given us a set of powerful mechanisms to write our programs. In this chapter we again extend our powers of expression through mechanisms that let us abstract operations and pass them around for later operations.

19.1 BEYOND THE DUALITY

The extension will require treating *operations* as if they were *objects*, which seems at first to contradict the basic duality, so far taken for granted, between these two notions:

- Our programs manipulate objects.
- They do so by applying operations to these objects.

The textual structure of our O-O programs also relies on this distinction: we divide programs into classes, each based on a type of objects, and each operation is attached, in the form of a routine, to one of these classes.

The two notions seem completely distinct: what the program can do (operations); what it can do it to (objects).

And yet it is sometimes interesting to treat an operation as an object or, more precisely, to define objects whose sole role is to describe an operation. We'll call such objects *agents*. This chapter studies them in detail, but it's not hard to get the basic idea. You can obtain a simple agent through the notation

`agent r`

an expression whose value is an agent representing the routine *r*. Because it's an expression you can assign it to a variable, as in

`a := agent r`

with *a* of the appropriate type.

What can you do with an agent? Well, it is associated with a routine or other feature, so you can use the agent to call the routine. With a denoting an agent after the above assignment, the call

$a.call ([x, y])$	[1]
-------------------	-----

will have the same effect as if you directly called

$r (x, y)$	[2]
------------	-----

for any applicable x and y . The feature *call* is applicable to all agents; it takes a single tuple, here $[x, y]$, as argument. (To understand the rest of this chapter you'll need to be familiar with tuples, a simple notion studied [earlier](#).)

← "[TUPLES](#)", 10.5, page 266.

Why use *call* on an agent, as in [1], when you could just call the routine directly as in [2]? Indeed if you know the routine r you need to call there is no point in going through an agent. But now assume you got a from another program element, for example as a routine argument. Then all you know is that a denotes a routine (and also, as we'll see, what kind of arguments that routine takes); but you don't know the routine itself — the original r .

This is indeed what agents give us: the ability to pass around objects that represent operations ready to be executed, with a complete separation between:

- *Agent definition*: the place in the software that defines an agent around a routine r , through **agent** r , and which of course must know about r .
- *Agent use*: any place in the software that receives an agent a and can apply features such as *call* to it, without knowing what routine it carries.

This mechanism has many different applications, of which we will now review some of the most important:

- *Iteration*: providing a general mechanism that applies an arbitrary operation to every item of a data structure.
- Numerical programming, as when computing the integral of a function over an interval; we may represent the function as an agent.
- Equipping an interactive application with an undo-redo mechanism.

Another area where agents play an important role is *event-driven design*, also known as Publish-Subscribe, particularly useful for graphical user interfaces; it is the subject of the next chapter.

We will compare agents with other techniques, based on previously studied mechanisms such as dynamic binding, which would also be available to address some of these applications; we will take a look at the mathematical basis, the fascinating theory of *lambda calculus*; and we will look at techniques available in languages other than Eiffel.

19.2 WHY OBJECTIFY OPERATIONS?

It's good first to understand why we need some kind of mechanism to treat operations as objects, and what we would do if we didn't have it. Here are four examples.

Iteration, integration, observation, undoing

First, **iteration**. We have got used, in our loops, to schemes that apply a certain *operation* to every element of a sequential structure such as a list. They look like this:

```

from start until after loop
    "Apply action to item"
    forth
end

```

[3]

← The second line is
pseudocode (see
page [110](#).)

In Traffic, we can apply this scheme to an instance of *ROUTE*, denoting an itinerary with a number of stops. We might want to print the names of all stops in order; to compute the total travel time (by adding the time from each stop to the next one); to produce a list of restaurants along the route (from a list of nearby restaurants available for each stop); and so on.

In each case the solution will look like [3]. We have a name for such schemes: iteration, already encountered in the discussion of data structures. You can use such an iteration scheme, for any given *action*, to produce a routine that applies *action* to every stop along a route.

Now assume that you do *not* want to write a new routine each time you need this scheme. Can we go up one notch in abstraction and simply write something like [3] in a routine where *action* is not hardwired any more, but just an argument? Then we could use that routine with different actions, and let it take care of the looping.

Iteration mechanisms will indeed enable us to provide routines such as *do_all* which you can call with an agent argument representing the action:

```

your_route.do_all (agent action)

```

The second example is from numerical mathematics: **integration**. Given a function $f(x: REAL): REAL$ defined over an interval $[a, b]$, algorithms exist (we'll see the basic one below) to compute a good approximation of the integral of f over that interval:

$$\int_a^b f(x) dx$$

The problem here is: can we have a general mechanism — say a feature *integral* from a reusable class *INTEGRATOR* — that we can apply to any existing routine *f* representing the mathematical function? Agents will indeed allow us to provide such a mechanism, and call it as

```
your_integrator.integral (agent f, a, b)
```

(with *your_integrator* of type *INTEGRATOR*.)

The third example is a preview of the next chapter: event-driven design. Assume some part of the system can trigger some events and other parts need to execute some operations whenever such an event occurs. For example the event could be a clock tick, happening whenever a certain time has elapsed; then a clock display module must update an image, another module needs to update the total time count, and so on. Each such “subscriber” module needs to register a certain action to be executed whenever such an event occurs. We’ll see an architecture enabling subscribers to achieve this simply through

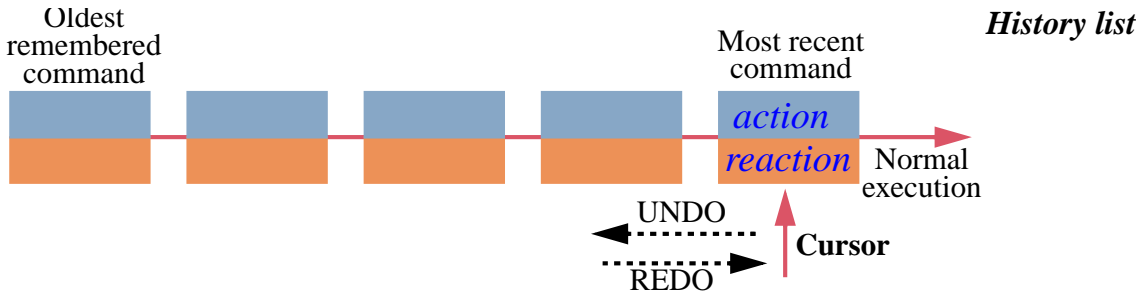
```
clock_tick.subscribe (agent some_routine)
```

where *clock_tick* represents the event type and *subscribe* is a general-purpose library feature.

The last example corresponds to a functionality widely needed in interactive systems: **undoing** an operation. While not widely acknowledged — I don’t know of any statues of its inventor — the humble CTRL-Z (or equivalent) is undeniably one of the milestones in human history, recognizing that we need to be saved from our own messing up. Even when we don’t actually mess up, we like to try out ideas, see what happens, and back up if the result is not to our liking.

The only really good undo-redo mechanism is one that lets you undo and redo not just the last operation but many. I probably don’t need much convincing when talking to you as a user of programs, but now think of how you would *write* a program with built-in undo-redo to any level.

The most radical technique involves representing all undoable-redoable actions as objects which you can put into a data structure, say *history*, which can be implemented as a list of agent pairs:



If these actions come from routines you never execute such a routine *r* directly but always through

```
execute (agent r, agent r_inverse)
```

where *execute* performs *call* (the mechanism for calling the routine associated with an agent, as previewed above) on its first argument, but also appends the object pair of its two arguments into the history list. Each pair in the history list contains two agents, one representing an action and the other — appearing as *reaction* in the figure — representing the reverse action; this assumes that for every routine *r* implementing a user command you also provide a routine *r_inverse* that cancels the action (otherwise you couldn't offer an undo-redo mechanism). Then if the user requests one or more “undo”, you perform

```
history.item.reaction  
history.back
```

as many times as needed (but of course not going further back than the first item); for a “redo” request after one or more “undo”, perform

```
history.forth  
history.item.action
```

(not going further than the last item).

A world without agents

We can't really understand agents in depth unless we ask ourselves how we would address the above problems if we didn't have a special mechanism.

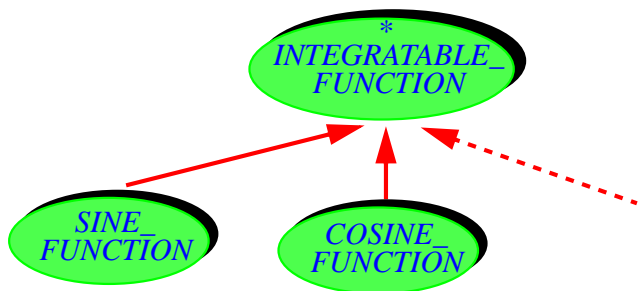
Can we find a solution at all? Of course we can. If what you need is an object wrapper around an action, it suffices to create that object yourself, devising the appropriate class. That will be the hurdle: defining new classes. Let's see the idea at work in the previous examples.

Integration is typical. To define the integration function *integral*, give it an argument of type *INTEGRATABLE_FUNCTION*. That's a deferred class which could look like:

```

note
  what: "Functions that can be integrated over finite intervals"
deferred class INTEGRATABLE_FUNCTION feature
  value (x: REAL): REAL
    -- Function's value at position x
  deferred
  end
end

```



*Classes for
mathematical
functions*

For each particular function to be integrated we have to write, as the figure suggests, a little effective class such as *SINE_FUNCTION* which simply provides the desired implementation of *value*:

```

value (x: REAL): REAL
  -- Value of this particular function for x
do
  Result := sine (x)    -- Or any other desired value
end

```

← *sine* and any other math routine must be obtained, e.g. through inheritance, from some math library class.

One could make the class *INTEGRATABLE_FUNCTION* more sophisticated — for example by introducing a query *defined* (*x*: *REAL*) — but this simple version suffices to understand the architectural issue.

Then to obtain the integral of the sine function over $[a, b]$, you declare a variable *sine_function*: *SINE_FUNCTION*, create the corresponding object, and call

```

your_integrator.integral (sine_function)

```


Internally, the function *integral* (*f*: *INTEGRATABLE_FUNCTION*) is easy to write: whenever it needs to evaluate the value of the function at a certain point *x*, it obtains it through *f.value(x)*. Note the role of dynamic binding: the run-time type of *f*, such as *SINE_FUNCTION*, determines which *value* feature to use. To make sure that you understand this scheme and review your understanding of fundamental O-O techniques it's a good idea to try to write *integral* yourself (don't worry if numerical programming is not your forte, as the exercise suggests a model for the actual algorithm):

Programming time!
An integration library without agents

Write a class *INTEGRATION* with a feature *integral* that computes the integral, over a finite interval, of a function passed as an argument of type *INTEGRATABLE_FUNCTION*. Devising the appropriate descendants to *INTEGRATABLE_FUNCTION*, apply your work to the computation of integrals of various sample functions.

For a simple integration algorithm, you can use the model of the agent-based version given below.

→ ["AGENTS FOR NUMERICAL PROGRAMMING", 19.4, page 473.](#)

This example is typical of how to take advantage of standard O-O techniques that you can use if you don't have agents. The ideas are easily transposed to all our other examples:

- For iteration, the deferred class will be *ITERATABLE_ACTION*; effective descendants provide specific versions of a procedure *call* describing one execution of the iterated operation.
- For observation (event-driven programming), the deferred class is *OBSERVER*; specific observer classes inherit from it and provide their own versions of the *update* procedure which publishers will call when triggering an event. What this describes is the exact principle of a well-known "design pattern", Observer, covered in the next chapter.
- For Undo-Redo, each command of the interactive system must be implemented by a command class providing two procedures: *execute* to perform the command, and *cancel* to undo the effect of the last *execute*. An instance of this class describes information resulting from one execution of the command and necessary to undo it later should this be requested; for example, in a text editor, an instance of *LINE_DELETION* has two fields, the text of the line being deleted and the position of that line in the text, so that the *cancel* procedure can re-insert a line deleted by *execute*. All such command classes inherit from a deferred class *COMMAND* where *execute* and *cancel* are deferred. The history list can then be implemented as, for example, a *LINKED_LIST [COMMAND]*.

→ ["THE OBSERVER PATTERN", 20.4, page 515.](#)

We may call this technique the “**Many-Little-Wrappers pattern**” because it uses dynamic binding based on writing a class, typically small, to wrap each variant of an operation.

The pattern works, but it has the obvious disadvantage suggested by its name: bloating the software with numerous small classes. There’s nothing wrong in principle with small classes, but a class should embody a significant abstraction, and having just one significant feature (such as *value* in the integration case) makes it suspicious. This suspicion is reinforced by the observation that in two of the examples (integration and iteration) the classes have only one significant feature (*value*, *call*). In particular, they have no attributes, and hence each needs only one instance (such as *sine_function*). A class with just one instance is known as a **singleton**, but here it’s not just that, since the single object has no fields. Each of the classes is essentially there to encapsulate a single routine. We may call them **Single-Song-Artist** classes.

Having to write many such wrappers uselessly complicates the software — in particular its inheritance structure, as we’ll see for the Observer pattern in the next chapter. In the end, it’s frustrating that we can’t directly use the *sine* function for integration, or a routine *print_stop_name* for route traversal. Why do we need a class? In an extreme case, the same mathematical operation could conceivably be amenable to integration *and* iteration *and* observation; we would then wrap it in three different ways!

Among our examples, the one case where the wrapping doesn’t come out as too artificial or bothersome is undo-redo, because the **COMMAND** abstraction seems warranted: it has two equally important routines, *execute* and *cancel*, so it’s at least a two-song artist; and descendant classes describe meaningful objects, with many different instances (for example every execution of a **LINE_DELETION** yields a new instance) characterized by meaningful fields (such as the specific text and position of the deleted line)

In the other cases, it seems hard to justify the Many-Little-Wrappers technique. We do need to wrap individual routines into objects, but we don’t want to have to do the wrapping ourselves; a language mechanism will do the job, and that’s exactly what agents are for. If *f* is a routine, you can get it gift-wrapped for free by just writing **agent f**; this gives you an object that has everything about *f*, including the ability to call *f* (through the procedure *call* applicable to all agents) whenever you need to, and for any applicable arguments. In all the cases cited including undo-redo, and many others, using agents is vastly superior to the Many-Little-Wrappers pattern. (We’ll review some alternative techniques at the end of this chapter.) So I hope you didn’t mind this little digression — discussing, in the chapter about agents, what to do *without* agents — since it should give us a much better appreciation of much we can benefit from a simple, built-in mechanism for dealing with actions through objects.

→ “**OTHER LANGUAGE CONSTRUCTS**”, 19.9, page 494.

19.3 AGENTS FOR ITERATION

Now that we see where agents fit and why we need them, let's go beyond the earlier overview and see the full details. The present section completes the iterator example; the next one deals with integration. The next chapter has a detailed agent-based solution to the problem of event observation.

Basic iterating schemes

A simple example from Traffic illustrates the use and definition of iterators through agents. Consider the notion of *ROUTE*. We can add to *ROUTE* a routine *do_at_every_stop* that takes an action as argument and applies it to every stop. This will make it possible to use

```
your_route.do_at_every_stop (agent print_name)
your_route.do_at_every_stop (agent append_restaurants)
...
your_route.do_at_every_stop (agent other_operation)
```

This simply assumes that *print_name*, *append_restaurants* and *other_operation* are routines, in this case, specifically, procedures, which take a *STOP* as argument.

How will *do_at_every_stop* achieve this? It abstracts the standard iteration scheme cited earlier in this chapter [3]:

← Page 459.

```
do_at_every_stop (action :...) is
  -- Apply action to every stop in this route.
  do
    from start until after loop
      action.call ([item])
    forth
  end
end
```

→ The type for *action* appears below (see page 467).

To trigger the associated routine, this uses *call*, a procedure available on all agents, whose effect is (as you may guess) to call the agent's routine, with the arguments given; more precisely, given as a single *tuple*, here *[item]*. A *tuple* is a sequence of values, written in square brackets; since *action* is intended to represent routines such as *print_name* that take one argument, the tuple used here, *[item]*, has just one element.

← "TUPLES", 10.5, page 266.

The effect of the highlighted call *action.call ([item])* [4] is exactly the same as that of a direct call to the corresponding routine, such as

```
print_name (item)
```

[5]

if the argument passed to *do_at_every_stop* was **agent** *print_name*, or

<i>append_restaurants</i> (<i>item</i>)	[6]
---	-----

if the argument was **agent** *append_restaurants* and so on. The difference with [4] is that within *do_at_every_stop* we do not know what actual routine *action* represents, so we can't use a direct call such as [5] or [6]; that's where we need an agent, represented here by *action*.

This technique is the basic mechanism for providing iterators in the EiffelBase library; we'll take a look below at the actual library implementation.

→ "*The anatomy of an iterator*", page 469.

Iterating for predicate calculus

An interesting application of iteration is to give us a direct implementation of the predicate calculus mechanisms: for all (\forall), there exists (\exists). Assume for example that you want to state that all elements of a certain array of integers *a*, of bounds *a.lower* and *a.upper*, are positive. In predicate calculus we have learned to express this as

$\forall s: a.lower .. a.upper \mid a [i] > 0$	[7]
--	-----

where *i..j* denotes the interval containing all values between *i* and *j* inclusive. Without agents you can use *all_positive* (*a*) if you write a function *all_positive* (*ia*: *ARRAY [INTEGER]*) which determines the result through a loop. But thanks to agents you don't need to write such a routine; just use

<i>(a.lower .. a.upper).for_all</i> (agent <i>is_positive</i>)	[8]
---	-----

which is very close to [7]. *|..|* is the operator alias of a function *interval* from class *INTEGER*, which yields a result of type *INTEGER_INTERVAL* — not a predefined concept but a normal library class, providing functions *for_all* and *there_exists* which take as argument an agent representing the test being “for-alled” or “there-existed” across the interval. In a short while we'll see similar functions available on lists and other sequential structures, which will also be the opportunity to clarify the signatures.

[8] still requires you to write a small function to test whether an integer is positive: *is_positive* (*n*: *INTEGER*): *BOOLEAN*. This is more reasonable than having something like *all_positive* for every such case. At the end of this chapter we'll see how to get rid of even *is_positive* by writing the needed agent “inline”, without having to introduce an explicit routine.

Exploration time!

You may wish to take a quick look now at the functions *for_all* and *there_exists* in *INTEGER_INTERVAL*. Don't get stuck with the type declarations (they are explained next), but make sure you understand the algorithms.

Agent types

In the declaration of *do_at_every_stop* we need to fill in the type of *action*, representing an agent. The actual declaration will be:

```
do_at_every_stop (action: PROCEDURE [ANY, TUPLE [G]]
... The rest as before [4] ...
```

PROCEDURE is a generic library class describing command (procedure) agents. It takes two generic parameters, representing type properties of the procedure *p* associated with the agent:

- The first denotes the class from which *p* comes, or an ancestor of that class. Since *ANY* is ancestor to all classes you can usually use *ANY*, as we do here with *do_at_every_stop*, since in an actual argument **agent** *p* corresponding to *action* we don't care what class *p* comes from.
- The second parameter is always a tuple type. The tuple component types correspond to the types of the arguments to *p*; here, since we expect procedures such as *print_name* and *append_restaurants* that take one argument of type *G* (the generic parameter of *LINEAR* and descendants, also serving as the type for *item* and representing the type of the items in the data structure), we use *TUPLE [G]*.

It is this choice of as the second parameter type for *PROCEDURE* that enables us in the body of *do_it_all* to call the associated routine *p*, whatever it is, with valid arguments, through the following line from [4]:

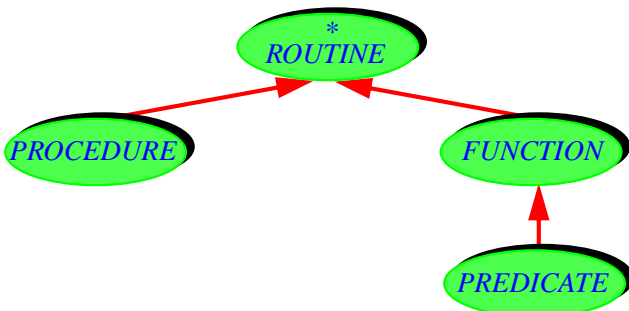
```
action.call (item)
```

[9]

Indeed, procedure *call* is declared in *PROCEDURE* (check the source for yourself in the library!) as taking an argument of type *OPEN*, the second generic parameter.

PROCEDURE covers agents associated with commands. It is part of a hierarchy of four classes in the Kernel Library:

Agent classes



FUNCTION is for agents denoting queries, except for those returning a result of type *BOOLEAN*, represented by *PREDICATE*. *FUNCTION* has a third generic parameter, representing the type of the function's result. *ROUTINE* is a deferred class, for representing arbitrary agents.

Here are the headers of the classes involved:

```
deferred class ROUTINE [BASE, OPEN → TUPLE]
class PROCEDURE [BASE, OPEN → TUPLE] inherit
  ROUTINE [BASE, OPEN]
class FUNCTION [BASE, OPEN → TUPLE, RES] inherit
  ROUTINE [BASE, OPEN]
class PREDICATE [BASE, OPEN → TUPLE] inherit
  FUNCTION [BASE, OPEN, BOOLEAN]
```

*In the actual class texts, the formal generic matters have longer names *BASE_TYPE*, *OPEN_ARGS* and *RESULT_TYPE* to avoid conflicts with programmer-chosen class names.*

The second parameter, *OPEN*, is constrained by *TUPLE*, so you can only use a tuple type — *TUPLE [G]* in the above example — as actual parameter. Given a feature *f*, the expression **agent** *f* is of a type derived from one of the above, depending on the nature of *f*: procedure, boolean query, other query.

For an agent representing procedures with two arguments of types *T* and *U*, use

```
PROCEDURE [C, TUPLE [T, U]]      -- C is often just ANY
```

and similarly for the other cases. For a routine with no arguments, the second actual parameter will be just *TUPLE*, as in *PROCEDURE [ANY, TUPLE]*.

Class *ROUTINE* declares:

```
call (v: OPEN)
  -- Call feature with all its operands, using v for the open operands.
```

allowing you to call the agent by passing an appropriate tuple as illustrated above [9]. If there are no arguments — the actual parameter for *OPEN* was just *TUPLE* —, you will pass to *call* an empty tuple [].

In addition, *FUNCTION* and *PREDICATE* have the feature

```
last_result: RES
  -- Function result returned by last call to call, if any
```

and, for convenience, the function *item* combining *call* and *last_result*:

```
item (v: like open_operands): RES
  -- Result of calling feature with all its operands,
  -- using v for the open operands.
  -- (Will call call.)
ensure
  set_by_call: Result = last_result
```

→ In cases so far the “open operands” are the arguments. For more, see “*OPEN OPERANDS*”, 19.5, page 475.

The anatomy of an iterator

Class *LINEAR* in EiffelBase, ancestor to all the list classes such as *LIST*, *LINKED_LIST* and others, describes any structure that can be traversed linearly. As such, it is the natural home for a set of iterator features:

- *do_all* applies a certain action in turn to all item of the structure, like our *do_at_every_stop* example.
- *do_if* applies it to all elements that satisfy a certain condition; there are also *do_while* and *do_until*.
- *for_all* tests whether a certain property (again represented by an agent) holds of all elements of a structure, and *there_exists* which tests whether it holds of at least one.

The arguments are:

- In the first two categories, *action* representing the action to be applied, of type *PROCEDURE [ANY, TUPLE [G]]*.
- In the last two categories, *test* representing a test, of type *PREDICATE [ANY, TUPLE [G]]*.

(*do_if*, *do_while* and *do_until* have both arguments.) As an example of use, assume a certain class has an integer attribute *sum* and the (trivial procedure

```
increase_sum (n: INTEGER)
    -- Add n to sum.
    do sum := sum + n ensure added: sum = old sum + n end
```

Then given a list *il*: *LIST [INTEGER]* the call

```
il.do_all (agent increase_sum)
```

will (after *sum := 0*) assign to *sum* the total of the values of *il*'s items.

Now for the internal picture:

Anatomy Lesson

As part of our regular series of examining *real* code in depth, we now take a look at the text of *do_all* in class *LINEAR [G]* from EiffelBase. You are encouraged afterwards to explore its companions such as *do_if* and *for_all*.

The routine's text, copy-pasted from the [library](#), appears at the top of the next page. After this discussion you can and should understand everything about it.

Version 5.7.

Procedure *do_all* takes a single argument, *action*, representing the agent to be iterated. The type *PROCEDURE [ANY, TUPLE [G]]* indicates that the agent's associated routine can come from an arbitrary class (as expressed *ANY*) and (as expressed by *TUPLE [G]*) should take one argument of type *G*, the formal generic parameter of the enclosing class.

```

do_all (action: PROCEDURE [ANY, TUPLE [G]]) is
  -- Apply action to every item.
  -- Semantics not guaranteed if action changes the structure;
  -- In such a case, apply iterator to clone of structure instead.
  local
    t: TUPLE [first: G]
    cs: CURSOR_STRUCTURE [G]
    c: CURSOR
  do
    cs ?= Current
    if cs /= Void then
      c := cs.cursor
    end
    create t
    from
      start
    until
      after
    loop
      t.first := item
      action.call (t)
      forth
    end
    if cs /= Void then
      cs.go_to (c)
    end
  end
end

```

Procedure
do_all in class
LINEAR

The header comment tells us “Semantics not guaranteed if *action* changes the structure”. The warning is important warning: havoc could result if *action* changes the data structure itself, for example by deleting an element. (An [exercise](#) asks you to try this for yourself if you have the nerve.) It’s OK for *action* to change the *contents* of objects in the structure; for example you can safely use *do_all* to add one to every element of a list of integers, through

→ “An iterator that shoots itself in the foot”, 19-E.5, page 500.

```
do_all (agent increment)
```

with

```
increment do item := item + 1 end
```

since this doesn’t modify the structure. If you *do* want to modify the structure, the last line of *do_all*’s header comment indicates that it is safe to iterate on a clone (duplicate) of the original structure; then *action* can modify the original without affecting the clone. For a clone of a structure *s*, simply use *s.cloned*.

The requirement stated by this header comment is legitimate: the notion of iterating on a data structure stops making sense if the structure itself changes as you are iterating on it. Still, as you may have reflected, it is regrettable that to enforce it we have to resort to exhortation through a header comment, rather than expressing it in the contract for the routine, in the form of a precondition on *action*. Contracts are currently not expressive enough to state such properties.

Let us now look at the routine body, which for convenience has been divided into three parts. We look first at “Part 2”, the core of the algorithm. This is an improved form of the basic iteration loop given above [4] as

```

from start until after loop
    action.call ([item])
    forth
end

```

[10]

Instead of a manifest tuple [*item*] the library version uses a local variable *t* representing a tuple, creates the tuple on initialization through **create** *t*, and instead of *action.call* ([*item*]) performs

```

t.first := item
action.call (t)

```

[11]

where *t* has been declared (see the **local** declarations) as a tuple with a component labeled *first*. These instructions fill the tuple *t* with *item*, yielding the same tuple value as in the manifest form [*item*], and call *call* with *t* as argument. The effect is the same; why the more complicated form? IT’s for efficiency: a manifest tuple expression such as [*item*] represents a tuple to be created with the components given, here just one, every time the expression is evaluated. This operation requires allocating memory and is expensive; but we don’t need it each time: once the tuple has been passed as argument to *call* the computation doesn’t need it any more. It will eventually be garbage-collected, so there is no long-term effect on memory usage, but it’s still a waste of computation time to create all these tuples for single use.

It’s like using a new paper cup each time you get yourself a coffee. If you are a regular coffee drinker, you will find it more convenient to get yourself a good cup once and for all. The library version does the same with *t*: create it once and reuse it ever after. It’s good to be aware of this technique:

Touch of Performance: **Saving on tuple allocation**

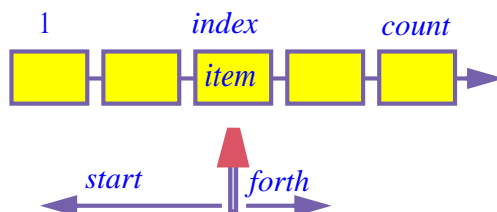
If you process a tuple in a loop or other program element executed repeatedly, and are sure that the computation will not need its value any more after that processing, do not use a manifest tuple but instead a tuple variable that you reset each time. This will save the overhead of repeated allocation.

Other “methodology” advice in this book warns you against practices that are potentially (but demonstrably) damaging for conceptual reasons, for example because they increase the likelihood of bugs, or make it harder to modify your software later. This one is different, since if you violate it there’s nothing wrong in your reasoning, just an efficiency problem. In a better world you should not have to worry about such aspects; a compiler would let you write [4], which is simpler and more obvious to the reader than [11], and after checking that the value of each manifest tuple is needed only once it would optimize the generated code as if you had written [11]. This would be better not only because it does the optimization for you but also — and in the end more importantly — because the burden of checking that the optimization is *correct* (that you don’t need the tuple any more) would fall on the compiler, not on you, avoiding errors. Eiffel compilers already perform many optimizations, some very clever, but not yet this one; so it’s something you must still handle yourself if efficiency is a concern.

A similar issue exists with strings. A manifest string — a string given explicitly by its characters listed in double quotes, such as “ABCDE” — denotes a new object, allocated anew each time the string is evaluated. In a loop, this is usually not what you want. In this case the language gives you an explicit variant that avoids repeated evaluation; write the string as **once** “ABCDE”. You can also treat the string as a declared “manifest attribute”: *my_string: STRING = “ABCDE”* (indeed it is generally better methodologically to give symbolic names to constants than use them explicitly in instructions; this facilitates change). With this approach the string is truly shared: any change to its characters will be retained for the next use.

It remains to understand the routine fragments labeled “Part 1” and “Part 2” above. They have a common purpose: making sure that the iterator leaves the structure in the state where it found it. As you know, many of our linear structures have cursors; the iteration in *do_all* moves the cursor by using *start* and *forth*. But other parts of the software may be using the list and, after moving the cursor to some position, may expect to find it in that same position later in the absence of an explicit operation to move it in-between *do_all* and other iterators must be good citizens: they can move the cursor while they operate, but they must restore it when they are done. “Part 1” saves the original cursor position; “Part 2” restores the cursor to the saved position.

This is only relevant for sequential structures that have a cursor, as is indeed the case with list classes. Such classes are descendants both of *LINEAR* and of a class *CURSOR_STRUCTURE*. “Part 1” finds out if the current object is an instance of that class by performing an assignment attempt to a variable *cs* of type *CURSOR_STRUCTURE*; if it succeeds, it records the current position into a variable *c* of type *CURSOR*, using the feature *cursor* of *CURSOR_STRUCTURE*. An instance of *CURSOR* is an abstract description of a cursor position, independent of the representation; the class has (as you can see by looking it up in EiffelStudio) descendants describing cursors specific to various structures, such as *LINKED_LIST_CURSOR*. Then “Part 2” brings the cursor to its initial position through the procedure *go_to* (*c: CURSOR*) of *CURSOR_STRUCTURE*.



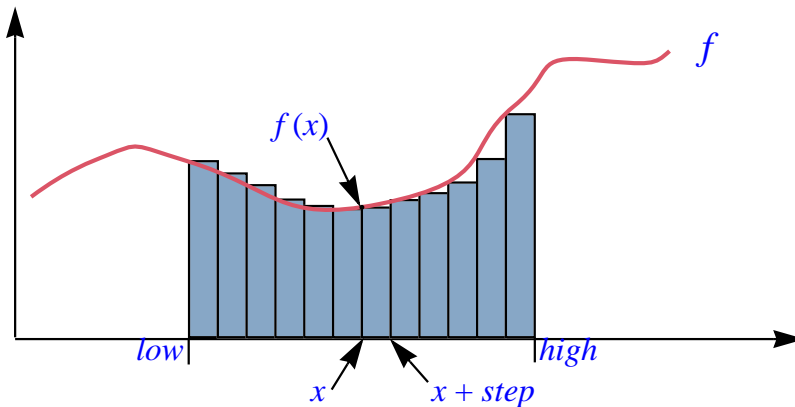
← From “[Cursor movement](#)”, page 272.

← See “[Assignment attempt](#)”, page 455. An exercise asks you to replace it by the newer language mechanism: “[Using Object Test](#)”, 19-E.4, page 500.

19.4 AGENTS FOR NUMERICAL PROGRAMMING

As the example of iterators shows, agents enable us to describe operations that manipulate other operations. One area in which such needs frequently arise is numerical programming; let's look at a typical example, integration.

The standard numerical technique, to integrate a real function f over a finite interval $[low, high]$, is to approximate the exact integral $\int_{low}^{high} f(x) dx$ by the sum of the areas of many small rectangles:



*Integration by
finite
approximation*

If all these rectangles have width $step$, the one starting at abscissa x has area $step * f(x)$. The approximation of the integral over an interval is the sum of such values for all x such that $low \leq x < high$.

What agents give us here is the possibility of writing an integration function *integral* not just for a specific function f — say the cosine function — but for any applicable function. The implementation of *integral* can be:

```

integral (f: FUNCTION [ANY, TUPLE [REAL], REAL] ; low, high: REAL)
  -- Approximation of integral of f over interval [low, high]
  local
    x: REAL ; i: INTEGER      -- See below about the role of i.
  do
    from x := low until x >= high loop
      Result := Result + f.item ([x])
      i := i + 1; x := low + i * step
    end
  end
end

```

→ “A numerical analysis note”, page 474.

The declaration of f indicates that it must be a function that takes an argument of type *REAL* and yields a result of type *REAL*. To evaluate f at point x we just use $f.item(x)$; as noted above, *item*, applicable to function agents, calls the function through *call* and returns the result of the call.

Function *integral* most likely belongs in a class *INTEGRATOR* describing objects in charge of performing integration operations on mathematical functions. If that's the design we retain, and *your_integrator* is of the corresponding type, then you will obtain the integral of a function f over an interval $[a, b]$ as the value of

`your_integrator.integral (agent f, a, b)`

Class *INTEGRATOR* is also where *step* should be declared as a *REAL* attribute, with an associated setter procedure so that clients can control the precision of the integration process. The class is more than a mere “wrapper” for *integral*; it describes a meaningful abstraction, “integration control”.

A numerical analysis note

Unrelated to the discussion of agents but important in practice is a numerical property of the algorithm used. The variable i is conceptually unnecessary: we could replace the last two instructions, which advance the loop (the last line before the two final **ends**), by just $x := x + step$. In pure mathematics this is indeed correct, but on a computer our *REAL* numbers are only approximations of the real “reals”; as we have seen, operations such as addition are only approximate, and if when they are carried out repeatedly the errors may accumulate. This would be the case with repeated $x := x + step$ operations: successive values of x could drift progressively from the exact mathematical value, which is $low + i * step$ at the i -th step. To avoid this drift, we recompute the value of x each time from the formula, so for each value we get at worst the error of a *single* addition and multiplication.

← “*Real*” numbers”,
page 319.

There's a general principle here:

Touch of Methodology: Computing with real numbers

In software that deals with computer representations of real numbers, be aware of the approximations involved, and devise the algorithms so that they will avoid *accumulation* of approximation errors.

A shorter form of this advice is “study numerical analysis” — the part of applied mathematics that deals with computing with actual numerical values (as opposed to *symbolic* computation), taking into account the properties and limitations of number representation and operations on actual computers.

19.5 OPEN OPERANDS

Sometimes you will need a bit more flexibility in using agents. A simple case is a variant of the last example where you still want to compute a function's integral for a certain argument, but the function has extra arguments:

$$\int_a^b g(u, x, v) dx$$

Variables u and v retain constant values during the integration — only the x axis is involved, as before — but they are still needed to evaluate g for every value of x . You could rely on the previous solution by obtaining the integral as

```
your_integrator.integral (agent g_extended , a, b) [12]
```

by defining an auxiliary function

```
g_extended (x: REAL)
  -- Same as g but with first and second arguments set to u and v
do
  Result := g (u, x, v)
end
```

assuming that u and v are attributes. This works, but it's tedious to write such auxiliary functions, especially if the pattern recurs. It will be even more unpleasant if u and v are local variables or formal arguments.

A function such as `g_extended` is just a wrapper, whose only purpose is to freeze some of the arguments of a function, turning it into a function of the remaining arguments only. This is needed so often that a special notation is appropriate. You can obtain the same effect as [12], without writing a wrapper function, through the expression

```
your_integrator.integral (agent g (u, ?, v), a, b)
```

The agent expression `agent g (u, ?, v)` denotes:

The one-argument function obtained from the three-argument function g by freezing its first and third arguments, to the values u and v respectively, and retaining only as a true argument the one at the second position, marked “?”.

More generally, in an agent expression you may use an argument list, corresponding to the signature of the underlying function, but in this list you may replace any of the arguments by a question mark **?**. These are known as **open arguments** to the agent, and the others — the ones given by normal values, like *u* and *v* above — as closed. Then the agent denotes a function of the open arguments only.

This means in particular that our first agent notation, **agent** *f*, is really just an abbreviation for

```
agent f (? , ? , ...)
```

with *all* arguments open. Of course in this case it's just as simple to use the shorter notation.

The notion of open argument increases the versatility of agents, saving the need for many auxiliary routines such as *g_extended*. As another example, let's vary a bit the *earlier* iteration scheme involving a list *il* of integers, assuming now that we have ← Page 469.

```
increase_sum_by_power (m, n: INTEGER, )
  -- Add to sum the value of m to the power n.
  do sum := sum + m ^ n ensure added: sum = old sum + m ^ n end
```

with *sum* now a *REAL* since that's the type the power operator \wedge returns. Then, starting with *sum* := 0.0, we can assign to *sum* the sum of the squares of all the elements of *il* through

```
il.do_all (agent increase_sum_by_power (? , 2)
```

To summarize:

Definition: open and closed operand

An operand of an agent is **closed** if it is specified in the agent's *definition*.

An operand is **closed** if it will be provided only in *calls* to the agent. In the agent definition, it is marked with a question mark **"?"**.

The "*definition*" of an agent is the expression that defines it, such as **agent** *f* (*a*, **?**). A "*call*" to an agent is an instruction (involving a call to *call*) or expression (involving a call to *item* for a *FUNCTION* agent) that calls its associated routine at run time. ← Page 468.

The last box sneakily introduced a new term, *operand*. So far we had been looking at open *arguments*. Why another notion? It's because sometimes what you will want to keep open includes not only arguments but also the **target** of a call. Consider again the earlier example involving routes and their stops:

```
your_route.do_all (agent print_name) [13]
```

where we can replace the specific procedure *do_at_every_stop* with merely *do_all*, since *ROUTE* is actually a descendant of *LINEAR [STOP]*. This assumes a procedure with the signature

```
print_name (s: STOP)
```

so that if it appears in a class *C* then **agent** *print_name* has the type *PROCEDURE [C, TUPLE [STOP]]*, matching the type for the formal argument of *do_all*. So *print_name* looks at *STOP* from the outside: it doesn't belong to this class but takes an argument of type *STOP*. This argument is the one that remains open since, as you know, [13] is really an abbreviation of

```
your_route.do_all (agent print_name (?)) [14]
```

A call to the agent, as executed by *do_all* — we saw exactly where it occurs: *action.call* (*t*), meaning *action.call* ([*item*]) for every *item* representing a *STOP* in the route — has the same effect as a direct call to the associated routine, of the form

← “Part 2” of the text of *do_all*, page 470.

```
print_name (your_route.item) [15]
```

where the highlighting emphasizes that the iterated action, corresponding to the ? in the agent, is passed as argument. The agent-based iteration scheme [13] is equivalent to a loop that would explicitly iterate through *your_route*, initializing the iteration through *your_route.start* and advancing it through *your_route.forth*, executing the call [15] at every step. Like any call in object-oriented programming, this call has a target, but it's implicit: the current object. (We can always make it explicit by writing the call as **Current.print_name** (*your_route.item*).)

Assume now, however, that we want to iterate not such an outside action but one given by a feature of class *STOP* itself. This appears quite legitimate. For example assume class *STOP* has a feature *close* that marks the current instance as a stop that's not in operation. If we want to close an entire line, we should iterate *close* over all its stops; but *close* takes no argument, since it's called just on its target, as in the typical call

```
some_stop.close
```

so that the action to be iterated, replacing [15], is

```
your_route.item.close
```

In this case it's the target, not an argument, that must be kept open in the argument to *do_all*, replacing *print_name (?)* in [14] (or the short form in [13]). At first we might think of writing that argument as something like *? .close* but this wouldn't work since it misses the type of the target; many classes may have a feature called *close*. We must specify the target type; the valid form for the example, illustrating the notation, is

```
your_route.do_all (agent {STOP}? .close) [16]
```

So now you see the need for the term *operand*: it covers all the values needed to execute a call — target and arguments.

For open arguments a plain “?” will generally do, since the types follow from the routine's signature — for example in [13] and [14] we know that *print_name* takes one argument of type *STOP* —, but the form “{TYPE} ?”, listing an explicit type, is also permitted.

All combinations of open and closed operands are permitted (assuming *f* with arguments and *g* without arguments in a class *C*):

- Everything closed: **agent** *f*(*a*, *b*, *c*), **agent** *g*.
- Target closed, all arguments (if any) open; this is the meaning of the abbreviated form with which we started, and which figures in most simple uses of agents: **agent** *f*, **agent** *g*. Also **agent** *f*(?, ?, ?) (means the same as **agent** *f*).
- Target closed, some arguments open, some closed: **agent** *f*(?, *b*, ?).
- Target open, some or all arguments closed: {*C*}?.**agent** *f*(?, *b*, ?).
- Everything open: {*C*}?.*f*, {*C*}?.*g*.

agent g fits both of the first two cases.

It's only thanks to these mechanisms that we can have a **single set of iterators** — *do_all*, *do_if*, *for_all* etc. — in *LINEAR* and all its descendants. Otherwise we would need a variant of each to iterate operations that work on their target, and another to iterate operation that work on their argument.

19.6 LAMBDA CALCULUS

We have now seen the basics of agents (and quite a few details as well); I hope you appreciate the power of expression of this mechanism and are already thinking of all kinds of wondrous applications.

There's actually one more level of flexibility, but before we get there I'd like to take you through a tour of the underlying mathematical ideas. The only way to understand what agents really are about — in particular, to get a deeper understanding of the concepts of “open” and “closed” operands as just studied — is to know the basics of *lambda calculus*.

It's a beautiful theory, developed in the 1930s before there were any computers; the discovery thirty years later that it provides a clear basis for many of the concepts of programming languages led to a revival of interest, and lambda calculus remains a fertile area of research.

What lambda calculus will give us is a theory of the notion of *function*, reduced to its essence: not any particular kind of function, such as the functions of trigonometry or real analysis with their specific properties, but the very idea of a function as a mechanism that takes certain arguments and yields a result. This is the *mathematical* notion of function, but since that's the concept behind functions and other features as we have them in programming the theory will give us new insights directly relevant for programming: what's the scope of a variable, what's the role of an argument, and how can we treat a function as if it were an object — the very goal that this chapter pursues by wrapping routines into agents.

Operations on functions

The basic idea is simple: a notation and transformation rules allowing us to play with functions as we play with other mathematical objects.

Given two numbers a and b , you can write combinations like $a + b$ or $\sin(a) + \cos(b)$; these use functions with well-defined signatures, for example

$\sin: REAL \rightarrow REAL$	-- Meaning: For any argument of type <i>REAL</i> ,
	-- <i>sin</i> yields a result of type <i>real</i>
"+": $[REAL \times REAL] \rightarrow REAL$	-- \times is cartesian product; brackets are for grouping

Can we do similar things with functions? Even in elementary mathematics there indeed are operators on functions: if f and g are functions with appropriate signatures, their composition, written $g \circ f$ or sometimes $f; g$ (the notation we'll use, because it retains the order of application), is the function h such that $h(x) = g(f(x))$ for any applicable argument x .

Lambda calculus will enable us to define many operators such as ";", whose arguments are functions.

We can continue up the ladder of abstraction. Composition, ";", can itself be viewed as a function: if f and g have — for some sets X, Y, Z — the signatures

$$\begin{array}{l} f: X \rightarrow Y \\ g: Y \rightarrow Z \end{array}$$

their composition $f; g$, called h above, has signature $X \rightarrow Z$. Now ";" as defined above can itself be defined as a function that given any two arguments such as f and g yields a result such as h . In other words it is a function of signature

$$";": [[X \rightarrow Y] \times [Y \rightarrow Z]] \rightarrow [X \rightarrow Z] \quad [17]$$

We can go on defining functions that operate on functions that themselves operate on functions and so on. Lambda calculus gives us a vocabulary and rules — in other words, a theory — for dealing with such functions at an arbitrary level.

Lambda expressions

First we need a simple notation for defining functions. We'll assume that we have at our disposal basic operations such as "+" on integers and reals; this is only for the sake of examples, since lambda calculus can be defined without reference to such existing mathematical theories. The symbol \triangleq will mean "is defined as". To define a function "square" of signature

$$\text{square}: REAL \rightarrow REAL$$

yielding the square of a number, we write a **lambda expression** as follows:

$$\text{square} \triangleq \lambda x : REAL \mid x * x \quad [18]$$

The right-hand side (after the \triangleq) is the lambda expression; it denotes the function that, for any x of type $REAL$, yields $x * x$.

The symbol λ , "lambda", is just a matter of convention but gives the whole approach its name. To introduce the value, the lambda calculus literature generally uses a dot \cdot , as in $\lambda x : REAL \cdot x * x$, but this just doesn't work in an object-oriented context where " \cdot " has another role, so we'll use a vertical bar $|$ instead.

This is reminiscent of how we define a routine in programming:

```
square (x: REAL): REAL [19]
  -- Square of x
do
  Result := x * x
end
```

with a more compact form in line with mathematical practice. The variable following the λ , here x , is known as a **bound variable** of the lambda expression and is similar to the formal arguments of a routine.

The choice of bound variable does not affect the informal meaning of the lambda expression. Just as we may choose the name y for the argument to the routine [19], without affecting the routine's meaning as long as we use y instead of x throughout its text, [18] denotes the same function as

$$\lambda y : REAL \mid y * y$$

This observation will be formalized below through the notion of *alpha-conversion*.

A lambda expression may have more than one bound variable, as long as all its variables have different names:

$$\begin{aligned} \lambda x, y : INTEGER \mid x + y & \quad \text{-- The addition function} \\ \lambda x : NATURAL, z : REAL \mid z^x & \quad \text{-- Notation when the types are different} \end{aligned}$$

What do lambda expressions buy us? At first sight, [18] states the same property as if we had just said that *square* is the function such that

$$\forall x : REAL \mid \text{square}(x) = x * x \quad [20]$$

but the difference is that [20] only *talks about* the function *square*, giving properties of its values for possible arguments, whereas [18] **defines** *square* as a mathematical object in its own right, in the same way that we can define the number π by giving its value.

One of the immediate benefits is to allow clear definitions of higher-order functions such as composition (signature given by [17]):

$$";" \triangleq \lambda f : X \rightarrow Y, g : Y \rightarrow Z \mid g(f(x))$$

X, Y, Z are assumed to be known sets. Since they are arbitrary, we could introduce a genericity mechanism for lambda expressions, as for classes, turning X, Y and Z here into formal generic parameters. This is not necessary for this short overview of lambda calculus.

In this example the source set in the signature, $[X \rightarrow Y] \times [Y \rightarrow Z]$, is a cartesian product; correspondingly, the lambda expression has two bound variables f and g .

Either an approximate value, of the exact value as a sequence limit or other math formula.

You will have noted that so far every definition of a function by a lambda expression has been preceded by a specification of the signature of the function; in addition, every bound variable is declared with its type (as in $f: X \rightarrow Y$), like a formal argument in a routine. It is also possible to omit the types entirely, with lambda expressions such as $\lambda f, g \mid g(f(x))$, yielding the variant of the approach known as **untyped lambda calculus**. Here we'll stick to the typed form, for the same reasons we use typing in programming with languages like Eiffel: readability, and avoiding errors.

Touch of Methodology: **Declaring the signature**

Whenever defining a function by a lambda expression, precede the definition by a declaration of the function's signature.

If the signature appears just before it's OK to omit the declarations of the bound variables, as in

";": $[[X \rightarrow Y] \times [Y \rightarrow Z]] \rightarrow [X \rightarrow Z]$
";" $\triangleq \lambda f, g \mid g(f(x))$

Currying

As an example of higher-order function that can be described through a lambda expression, consider *currying*.

Currying — so named in honor of the American mathematician Haskell Curry, one of the founders of the theory known as *combinatory logic* of which lambda calculus is a part — allows us, without loss of generality, to work only with functions of just one argument.

Touch of Notation: **Brackets and parentheses**

In ordinary mathematical notation parentheses serve both for grouping and for function application, as in $f(a * (b + c))$ (innermost for grouping, outermost for application). This would be very confusing in a discussion of operators on functions.

In the present discussion, parentheses are **only for function application**; grouping uses square brackets. So

$[f; g](a * [b + c])$

is the application of the function $f; g$ (the composition of f and g) to an argument that is the product of a and $b + c$.

When we are given a function, it often has two arguments — like “;” as just seen, (on functions) and “+” (on reals) — or more. Consider such a function for given X, Y, Z :

$$f: [X \times Y] \rightarrow Z$$

From f we can define a function f' with signature

$$f': X \rightarrow [Y \rightarrow Z]$$

as

$$f' \triangleq \lambda x: X \mid [\lambda y: Y \mid f(x, y)] \quad [21]$$

What does this mean? Unlike f , function f' takes only one argument, of type X ; also unlike f it doesn't directly yield a result of type Z . Instead, for any argument x it yields a function — the one highlighted above, let's call it g — which *itself* takes an argument y of type Y , then yields a result of type Z . This result $g(y)$ is $f(x, y)$: the same as if we had directly applied f to *two* arguments.

We say that f' is the **curried** version of f . Currying a two-argument function means turning it into a one-argument function, related to the original by [21]. Another way of expressing this is to say that to curry a function is to *specialize* it on its first argument. This leaves a function of one argument.

So if *add* is the addition operation on integers (which we may write as $add \triangleq \lambda x, y: INTEGER \mid x + y$), then *curry* (*add*) is the function

$$add' \triangleq \lambda x: INTEGER \mid [\lambda y: INTEGER \mid x + y]$$

so that $add'(1)$, for example, is $\lambda y: INTEGER \mid 1 + x$: the “increment” function, adding one to any given integer.

The correspondence between a two-argument function f and its curried version (called f' above) is one-to-one: informally, we don't lose any information by specializing f on its first argument, since the effect of the second argument is embodied in the argument of f' .

In fact it is interesting — and an example of the expressive power of lambda notation — to express this correspondence between f and f' explicitly, by introducing currying itself as a function, say *curry*, defined by a lambda expression. For given X, Y, Z its signature is

$$curry: [[X \times Y] \rightarrow Z] \rightarrow [X \rightarrow [Y \rightarrow Z]]$$

and its value:

$$curry \triangleq \lambda f: [X \times Y] \rightarrow Z \mid [\lambda x: X \mid [\lambda y: Y \mid f(x, y)]]$$

You should similarly define the inverse function, yielding f from f' .

Generalized currying

Although our basic examples all curry a two-argument function on its first argument, it's easy to generalize the concept: you can curry any function of n arguments ($n \geq 1$) on any choice of m arguments ($1 \leq m \leq n$) simply by setting values for these arguments. This yields a function of the remaining $n - m$ arguments, representing a specialized version of the original function, also known as a **partial evaluation**. If $m = n$, you get a constant function.

Currying in practice

As an example of what currying represents in practice, consider the difference between *compilation* and *interpretation*. If we have an interpreter for a programming language, we may view it abstractly as a function of signature

$$\text{interpreter: Program} \times \text{Input} \rightarrow \text{Output}$$

where *Program* is the set of all correct programs in the language, *Input* the set of possible inputs and *Output* the set of possible outputs. (This is a simplified but not incorrect view of what programs are about.) Now a *compiler* produces,

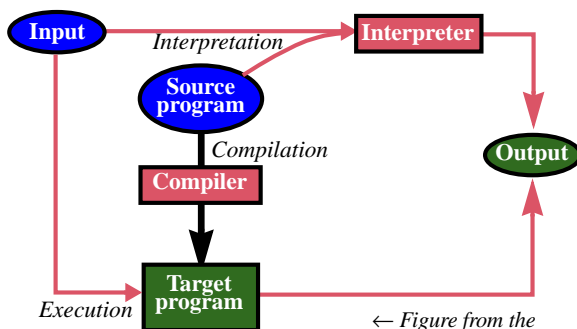
from the source program, a machine code program; because we have a mechanism — the hardware — to execute such programs without further effort on our part, we may consider them to be members of the set

$$\text{Machine_program} \triangleq \text{Input} \rightarrow \text{Output}$$

A compiler generates such a machine code program from a source program, so it is abstractly a function of signature

$$\text{compiler: Program} \rightarrow [\text{Input} \rightarrow \text{Output}]$$

When we have two possible execution mechanisms for the same programming language it is *very* important that they implement exactly the same semantics. (This is critical for example in EiffelStudio, where you typically go back and forth between the fully compiled, fully optimized *finalized* form of compilation and the fast incremental recompilation or *Melting Ice*, which is mostly interpreted; certainly you want your finalized code as delivered to your customer to produce — just faster — *exactly* the same result as the Melting Ice version.) Stating that:



← Figure from the discussion of compilation, page 355.

$compiler = curry(interpreter)$

captures this consistency requirement concisely and elegantly.

The notion of currying is particularly relevant for an object-oriented programmer. At the center of O-O style of programming appears a particular style that makes every operation relative to an object, as in the standard call

$x.f(args)$

(but also in an “unqualified” call $f(args)$, which is the same operation applied to the current object, and which you may indeed also write $Current.f(args)$). O-O programmer never really use the idiom “Apply this operation to *those* objects out there”, ubiquitous in other forms of programming. Instead it’s always “Apply this operation to *this* object x over here — oh, and by the way, you might need a couple arguments, here’s $args$ for you”. In the end you can express with one style what you would with the other, but the consequences of the object-oriented style on the structure of programs are profound: it makes the notion of class possible, as a form of both type and module, and opens the way to inheritance.

In the end, it comes down to the concepts of this section. Object-oriented programming is curried programming.

The calculus

What we have seen so far of lambda calculus is lambda expressions: a notation, providing useful insights, but not a calculus. The calculus, relying on that notation, provides a fascinating theory of functions and the operations on them; the theory is of course beyond the scope of this book but it helps to be familiar with the basic notions.

It turns out that lambda calculus can model the general notion of *computation* through two basic operations on lambda expressions: alpha-conversion and beta-reduction (also written α - and β -).

To define these notions we need to distinguish between two kinds of occurrence of a variable in a lambda expression: bound and free. As you remember we say that x, y, \dots are the *bound variables* of a lambda expression $\lambda x: X, y: Y, \dots \mid e$. Then an **occurrence** of a variable a in such an expression is **bound** if either:

- a is one of the bound variables (x, y, \dots) .
- The occurrence is (recursively) a bound occurrence of a in e .

This is an example of **recursive definition** as discussed in an [earlier](#) chapter.

The notion immediately generalizes to a non-lambda expression *exp*: an occurrence is bound in *exp* if it is bound in one of its lambda subexpressions. For example in $[f; g] (\lambda a : \text{INTEGER} \mid a + f(b))$ the occurrence of *a* is bound, but not those of *f*, *g* and *b*.

An occurrence that is not bound, such as those of *f*, *g* and *b* in this example, is **free**. As another example, in

$$\lambda x : \text{INTEGER} \mid [\lambda y : \text{INTEGER} \mid x + y + z]$$

the occurrences of *x* and *y* are bound, but the occurrence of *z* is free. Informally this means that *x* and *y* are names local to the expression, but *z* must be defined outside of it. This is exactly what we get in programming: in

$$f(x, y : \text{INTEGER}) : \text{INTEGER} \text{ do } \mathbf{Result} := x + y + z \mathbf{end}$$

x and *y* are formal arguments, meaning that they are just conventional names used to define the function, and any other names would work if they don't conflict with each other and with names from the enclosing class; but *z* must come from the context. In practice it should be a feature (specifically a query: attribute or function) of the class.

We say that *x* “occurs bound” in an expression *e* if it has at least one bound occurrence in *e*, and that it “occurs free” if it has at least one free occurrence.

The other basic notion is **substitution**:

Definition: variable substitution

Let *exp* be an expression, *x* a variable and *e* another expression. Then

$$exp [x := e]$$

denotes the expression obtained from *exp* by replacing (substituting) every free occurrence of *x* by *e*.

For example, if *exp* is

$$\lambda z : \text{INTEGER} \mid x + y + z * x$$

and *e* is *sin* (*x*), then $exp [x := e]$ is $\lambda z : \text{INTEGER} \mid \text{sin}(x) + y + z * \text{sin}(x)$. As this example indicates, it is possible for *e* to contain occurrences of *x*.

We only substitute free occurrences: if *exp* is

$$\lambda x, z : \text{INTEGER} \mid x + y + z * x$$

(the same as before except that x is now bound), then $exp [x := e]$ is identical to exp since we don't substitute the bound occurrences of x . But if exp is

$$\lambda y : \text{INTEGER} \mid f(x, [\lambda x : \text{INTEGER} \mid x + y])$$

we will substitute the first (highlighted) occurrence of x , which is free, but not the bound variable x in the innermost lambda expression, where this would make no sense because in that subexpression x is just an arbitrary name; $\lambda z : \text{INTEGER} \mid z + y$, where x doesn't appear, denotes (informally) exactly the same function. Alpha conversion will make this clear.

Let's actually begin with **beta reduction** because it is the central rule that captures the essence of lambda notation. Beta reduction enables us to get rid of a bound variable (and hence, if it's the only one, of a λ) by transforming

$$[\lambda x : X \mid exp] (e)$$

into

$$exp [x := e]$$

if **no free variable of e occurs bound in exp** . What this gives us is the notion of applying a function to actual arguments: since $\lambda x : X \mid exp$ stands for the function that yields exp as a function of x , applying it to e should stand for exp with every free occurrence of x replaced by e . For example, writing $e \xrightarrow{\beta} f$ for “beta-reduction transforms e into f ”:

→ A slightly less restrictive condition will do; see the exercise “Beta-reduction condition”, 19-E.7, page 500.

$$\begin{aligned} [\lambda x : X \mid x + y] (z) &\xrightarrow{\beta} z + y \\ [\lambda x : X \mid x + y] (y) &\xrightarrow{\beta} y + y \\ [\lambda x : X \mid x + y] (x) &\xrightarrow{\beta} x + y \\ [\lambda x : X \mid z + y] (e) &\xrightarrow{\beta} z + y \end{aligned}$$

In the last example, the bound variable x is not actually used in exp ; we may view the lambda expression as representing a constant function of x . So no substitution occurs when we apply the expression to an arbitrary argument e ; the lambda abstraction just disappears.

As the second and third examples show, having e use variables that occur in exp does not prevent beta reduction as long as these occurrences are not bound. This restriction does not rule out the third example, because exp is $x + y$, where x does not occur bound: it only occurs bound in the full enclosing lambda expression $\lambda x: X \mid x + y$. The restriction would only prevent beta-reduction in an expression like

$$[\lambda x: X \mid [\lambda y: Y \mid x + y]] (y) \quad [22]$$

where beta-reduction would yield $\lambda y: Y \mid y + y$, which incorrectly confuses y with the bound variable — “incorrectly” in light of the informal intent of the lambda expressions involved.

Does this mean we can never — through beta reduction — simplify $[\lambda x: X \mid exp] (e)$ if we are unfortunate enough that one of the free variables of e has been chosen as bound variable for some subexpression of exp ? This would of course be regrettable since bound variables are just arbitrary names. If we replace [22] by

$$[\lambda x: X \mid [\lambda z: Y \mid x + z]] (y)$$

beta-reduction becomes possible, yielding $\lambda z: Y \mid y + z$; but that was just a change of bound variable, not affecting the informal understanding of the underlying function. For this we need **alpha-conversion**, the rule allowing us to make such a harmless name change, as you would do in programming when you choose a new name for a variable or formal argument, for example to remove a conflict with the name of an attribute of the enclosing class.

Some programming languages allow such conflicts, with the convention that the most local name takes precedence; Eiffel prohibits them, for simplicity and readability, and to avoid bugs; so if a conflict arises you must find a new name.

Given a variable y , alpha-conversion transforms a lambda expression

$$\lambda x: X, \dots \mid exp,$$

in which y has **neither free nor bound occurrences**, into

$$\lambda y: X, \dots \mid exp [x := y]$$

→ Again the condition is, for simplicity, stronger than needed; see the exercise “Alpha-conversion condition”, 19-E.8, page 500.

The condition on y prohibits us from replacing x by y in either of

$$\lambda x: X \mid x + y$$

[23]

$$\lambda y: X \mid x + y$$

[24]

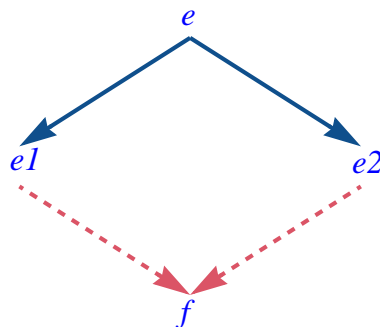
since the resulting expression $\lambda y: X \mid y + y$ would, in both cases, lose the semantics of the original expression:

- [23] represents a function of one argument, which adds y to this argument, y being a free variable, that is to say a value contributed by the context of the expression (for example an enclosing expression). In contrast $\lambda y: X \mid y + y$ is a function of one variable — locally usurping the name y —, which simply adds this variable to itself.
- In [24] y is bound, but then alpha-conversion would merge it with the free variable x .

The last observation indicates that the requirement on y as stated above is in fact stronger than it needs to be: we don't need to ban alpha-conversion if y if it has any bound occurrences in exp , only if any of these occurrences appears in a context where x is also bound.

→ Exercise [19-E.8](#).

Alpha-conversion and beta-reduction provide the basis for a full-fledged **theory of computation**, which describes any computation as a sequence of such transformations of (possibly complex) lambda expressions. A key consistency property of this theory is the *Church-Rosser Theorem*, stating that if from a given lambda expression e two separate sequences of transformations yield different expressions $e1$ and $e2$, then there exist two other sequences that transform $e1$ and $e2$, respectively into some common expression f (see the illustration). This means that if several transformations are possible on any particular expression, it doesn't matter which one you choose to apply first, as you'll eventually get to the same “canonical” result f .



**Church-Rosser
property**

→ Sequences of transformations (α , β or a mix)

Lambda calculus and agents

I hope that as you were reading about lambda calculus you started to make the connection with the concepts of the preceding sections.

The routine as we knew it until the present chapter was a program structuring construct, but couldn't join the games of program execution, along with references, basic objects (integers etc.) and more complex objects. To use a favorite expression of the functional programming, routines are not *first-class citizens* of program semantics. It's like functions in mathematics if we don't have a framework such as lambda calculus to define functions as values and manipulate them through expressions.

As lambda calculus turns functions into first-class citizens of the mathematical world, so do agents set routines loose, wrapped in objects, to act and be acted upon during execution.

Agents or not, routines are of course already a form of lambda expression, and routine call is the equivalent of beta-reduction. But it all has to be planned statically, with calls such as $f(a, b)$ where f is specified explicitly. Object-oriented programming introduces a first element of dynamism thanks to dynamic binding, allowing f to have several variants and the selection between them to be performed anew for each call $x.f(a, b)$ on the basis of the type of the object attached to x . This dynamic element is what enables us to solve some of the problems of this chapter through the Many-Little-Wrappers pattern (with the limitations that we've seen) but the choice it offers is only between a set of variants specified in advance. With agents, beta-reduction becomes a completely dynamic operation: $a.call([a, b])$ where — aside from the signature — we don't have statically to know anything about the routine that the agent a represents.

The concepts of lambda calculus help us understand the nature of “open” and “closed” operands. They correspond in agents to bound and free variables in lambda expressions.

- In $e \triangleq \lambda x : INTEGER \mid x + y$, the bound variable x represents an argument to be provided at the time of beta-reduction; the free variable y comes from the environment, typically an enclosing expression (it will remain unevaluated in a beta-reduction of e).
- In **agent** $f(?, y)$, the open argument will provided at the time of a call; the closed argument y is provided at the time of definition.

You may also have noted that having closed arguments in an agent is essentially to *curry* the routine on these arguments (taking currying in its general form applied to any m of n arguments). In the various dynamic forms that we can produce from a routine:

- **agent** $\{C\}.f$ is a dynamic version of the full f , faithful to the signature of the original.

← “Generalized currying”, page 484.

- At the other extreme, **agent** $f(a, b)$, with all operands closed, is the completely curried version; you call it (if the value of a is that agent) as $a.call()$ with no arguments. This, by the way, reminds us of one of the differences between mathematics and programming: we noted earlier that a math function curried on all its arguments is a constant function, but successive calls to $a.call()$ need not produce the same effect, because even if a hasn't changed the surrounding objects may have.
- In-between, an agent with some operands closed and some open, as in **agent** $a.f(x, y)$ is similar to a function curried on the closed operands.

In one respect, agents as seen so far are less general than lambda expressions. To use **agent** $a.f(x, y)$ or any of the other variants we must assume there's a function f to build on. It's as if in lambda expressions $\lambda x, \dots | exp$ we restricted exp to be of the form $f(args)$ where f is a function and the $args$ may involve some bound variables such as x (open arguments of the agent) and some free ones (closed arguments). We are now going to see how to remove that restriction and allow for agents the equivalent of arbitrary exp expressions.

19.7 INLINE AGENTS

The study of lambda calculus suggests a generalization of the basic agent mechanism, giving us flexibility beyond what we have already gained through the introduction of open and closed arguments.

The agents that we have used so far all proceed from an existing routine. But sometimes you want an agent and don't need a routine. You just want to define some computation or property to be passed along as an agent, and it's unpleasant to have to add a routine to the enclosing class just for that purpose; the routine may not be an interesting feature for the class, and will just make it seem more complex. **Inline** agents will let you define an agent without burdening the class.

The need often arises in writing contracts — all kinds: preconditions, postconditions, class invariants. For example the invariant of a class could specify that all the elements of a certain array of integers a are positive. We already know how to state this thanks to the class `INTEGER_INTERVAL` and the operator function `|..|` which, for any two integers a and b , enables us to express the interval $a |..| b$. We saw how to state the needed condition, equivalent to $\forall s: a.lower .. a.upper | a[s] > 0$:

$(a.lower |..| a.upper).for_all(\text{agent } is_positive)$

[25]

← “Iterating for predicate calculus”, page 466.

← Same as [8], page 466.

This requires writing a little function for the occasion:

```
is_positive (n: INTEGER): BOOLEAN [26]
  -- Is n greater than zero?
do
  Result := (n > 0)
ensure
  definition: Result = (n > 0)
end
```

That's a bit of a nuisance. Not so much the writing of the code: if you're worried about saving keystrokes you could get rid of the header comment and the postcondition, but you should not as this is not the problem (any useful algorithm, however local its use, should be properly designed and documented). The real issue arises if you only need *is_positive* for expressing the above property [25], for example as a clause in a class invariant. You are then encumbering the class with a feature that doesn't really belong to the corresponding data abstraction. This is particularly unpleasant if you have many such properties, as will be the case if you try to write precise and extensive contracts. True, you need not export these features, but they become part of the class anyway. It would be better to express the relevant properties or computations just at the place they are needed, with no visibility outside of that context.

Inline agents fit right here. An inline agent, as the name suggests, is a routine-like declaration yielding an agent — nothing else, no routine of the class — and declared where the agent is needed. The syntax is straightforward as illustrated by the rewriting of the last example; we merge [26] into [25], yielding

```
(a.lower |..a.upper).for_all
  (agent (n: INTEGER): BOOLEAN [27]
    -- Is n greater than zero?
    do
      Result := (n > 0)
    ensure
      definition: Result = (n > 0)
    end)
```

Starting with the second line it's the same as the earlier declaration; the only difference is that there is no routine name (*is_positive*) any more. That's indeed what characterizes an inline agent: it is an **anonymous routine**.

As illustrated, the syntax of an inline indeed that of a routine declaration, with the routine name replaced by the keyword **agent**. You can have use all the components that would be applicable to a routine, such as pre- and postconditions, or a **local** clause to give the agents its own local variables. Their names must be different from those of features of the class and local variables of the enclosing routines; this is different from the convention for lambda expressions (where inner bindings simply take precedence over outer ones), but is meant to avoid any confusion; as noted, names are not a scarce resource — or, put differently, you should take care of your own alpha-conversions.

Even though there can be no name conflicts with local variables of the enclosing routines, you may *not* use them directly in the agent. In the rare case you need them, you will have to pass them as arguments to the agent.

Illustrated above for predicates, the inline agent mechanism is useful for procedures and functions of any signature.

This mechanism completes our panoply of agent mechanisms, providing a major boost to the expressiveness of our object-oriented programs. In the next chapter we'll see a major application of this mechanism, addressing in an elegant way the “observation” problem sketched earlier.

19.8 A PROBLEM WITH AGENT TYPES

Before closing off on the agent mechanism it is useful to mention a limitation of the mechanism. You are unlikely to encounter the problem except if you start doing strange things with agents, but it exists and — in this book's spirit of rational analysis and open discussion — it's proper to mention it.

The problem involves type properties of tuples and agents. The general rule is that *TUPLE* [*T*, *U*, *V*] (for example) conforms to *TUPLE* [*T*, *U*], *TUPLE* [*T*] and plain *TUPLE*. It follows from the definition of *TUPLE* [*T*, *U*, *V*] as denoting sequences of three *or more* elements of which the first three are of the given types. It's in particular thanks to this rule that we can have a suitable generic parameter signature for *ROUTINE* and its descendants: *ROUTINE* [*BASE*, *OPEN* → *TUPLE*]. Since any tuple type conforms to *TUPLE*, we can freely use *PROCEDURE* [*ANY*, *TUPLE*] but also *PROCEDURE* [*ANY*, *TUPLE* [*T*]], *PROCEDURE* [*ANY*, *TUPLE* [*T*, *U*]] and so on.

← “*TUPLES*”, 10.5, page 266.

But now consider a routine expecting an agent of such a type

```
r (a: PROCEDURE [ANY, TUPLE [INTEGER] ]) do ... end
```

We think of *a*, as described earlier, as representing a procedure, from any class, expecting an integer argument. But in fact its type describes procedures of one *or more* arguments, the first being an integer. So it is in fact valid to write

```
r (agent (i, j: INTEGER do print (i + j) end))
```

using as actual argument for *a* a procedure with *two* arguments. Now *r* will most likely need to call *a*:

```
a.call ([n])
```

for some integer value *n*. But that's only *one* integer value; to execute the procedure — specifically, to print *i + j* — *r* needs another one, corresponding to *j*. The run-time result is undefined; the program might crash, or (generally worse) use an arbitrary value.

There is a simple way to avoid this: *ROUTINE* and its descendants provide a query

```
valid_arguments (args: OPEN)
```

which enables you, prior to calling *call*, to find out if an argument tuple is appropriate for the given arguments. Use it if you have a doubt. But it's not a really satisfactory long-term solution, first because it's tedious to prefix every *call* such a check, but more importantly because run time is too late for such a check: a mismatch should be detected at compile time. This is a language problem to which several solutions have been proposed. One will undoubtedly be available in the near future, but in the meantime just make sure, when assigning or passing an agent expression, that the number of arguments matches the signature exactly.

19.9 OTHER LANGUAGE CONSTRUCTS

At the beginning of this chapter we saw that a number of situations call for the possibility of passing around data — objects, in an O-O framework — that wrap computations. The agent mechanism addresses that need effectively.

← “[WHY OBJECTIFY OPERATIONS?](#)”,
19.2, page 459.

Not all programming languages, however, have such a construct. In fact, of languages commonly used in industry, only Eiffel, Smalltalk and C# have something like it (with significant differences in the details). So it's interesting to review briefly what solutions are available depending on the kind of languages you may have to use.

There are basically four approaches:

- A mechanism supporting lambda expressions, such as agents.
- Routines as arguments to other routines.
- Function pointers.
- In object-oriented programming, the Many-Little-Wrappers pattern.

Agent-like mechanisms

Agents as we have studied them in this chapter is a form of the first approach. **C#** offers *delegates*, which pursue the same aim. They are more complicated to use than agents since you must declare a specific type for each delegate. At the price of that complication, you will get the same effect.

Smalltalk has a notion of *block*, a segment of code that can be passed around as an object. Again, you get the same effect. Note that Smalltalk is an **untyped** language, meaning that there is no way to check at compile time that blocks will be used with the proper arguments; a mismatch will result in a run-time error.

Functional languages typically support the ability to treat functions (their routines) as data. This was already the case with the original Lisp, where an expression of the form

```
(defun f (x y) (“expression involving x and y”))
```

Lisp syntax.

defines **f** as a function of two arguments. Then you can use **f** as argument to another function, for example in

```
(curry f)
```

Lisp syntax.

where **curry** itself can be defined in Lisp. The language was indeed defined explicitly on the basis of (untyped) lambda calculus, so it is not surprising that much of what we have seen in this chapter can be done fairly naturally, with the qualifications that:

- Lisp is not object-oriented, so there is no notion of class, inheritance etc. in the basic language, although O-O versions of Lisp do exist.
- Lisp was originally untyped, so you will not get the benefits of static type checking, although here too the situation depends on which variant you use, as some of them are statically typed.

Functional languages, whether or not following lambda calculus as closely as Lisp does, generally provide similar capabilities. So for this entire class of languages you can use many of the techniques you have learned in this chapter.

The term **closure** is often used to denote expressions representing routines which can be passed around as data even though they may need to access global variables.

Routines as arguments

A number of programming languages allow you to pass a routine as argument to another routine, with a syntax such as

```
integral (f: function (x: REAL): REAL ; a, b: REAL): REAL
```

Not the exact syntax of a specific language.

and some appropriate notation to call such a routine argument, here `f`, from within the routine. You can then pass as actual argument a routine with a matching signature, as in `integral (sine, 0, 1)`.

This solution has the following limitations as compared to agents or closures:

- “Routine” is a special argument type which doesn’t generally fit well in the type system of the language.
- Typically, information about a routine is not a normal value (like an agent or a closure) and hence cannot for example be assigned to a variable (for which, because of the previous point, it would be hard to declare a type); it can only be used as argument to a routine.
- Because there is no proper type system, it is generally not possible or at least not simple to move up in abstraction and define functions such as composition or currying.
- All you can do on a routine argument is to call it. In contrast, agents are full-fledged objects whose features provide information on the encapsulated routine.
- Difficult issues arise when routines access global variables; they affect the compiler writer but also, to some extent, the programmer.
- The approach doesn’t fit too well with an object-oriented scheme, since it uses data other than objects.

The approach, however, fills many of the basic needs and has been used successfully in non-O-O languages, going as far back as Fortran and continuing with Pascal and several of its successors.

Function pointers

Computers, as you know, use memory to store not only the objects but the programs. At run time, a particular routine resides at a particular address, and it’s possible to transfer execution to the code at that address. If there’s a way for the program to denote that address, and a mechanism to say “execute routine at address *a*, then return and continue”, you can treat routine addresses as data through which to call the corresponding routines.

← *“The stored-program computer”, page 12.*

At the machine level this technique is what makes all the others possible:

- When you use a routine as argument to another routine, what the compiler will actually pass is the routine’s address.
- An agent object will internally contain — although not in a field that your program can directly access — is the address of the associated routine.
- Dynamic binding, necessary for the Many-Little-Wrappers pattern, assumes the run-time ability to access a routine known through its address, stored in some data structure associated with an object type.

All these techniques, however, hide the physical routine address under one or more layers of abstraction, enabling programmers to think in high-level terms: routines, agents, objects.

C and C++ let you pass the name of a function (the only kind of routine, procedures being treated as functions with a “void” result type) as actual argument, or assign it to a variable. Then if x is the corresponding formal argument or variable, you can call the original function through

`(*f) (args)`

When declaring a formal argument representing a function you can specify the full signature, known as a *prototype*, so that an actual argument that doesn’t match the signature will be rejected at compile time. This technique then becomes the same as the previous one (“routines as arguments”). Providing the signature is, however, not compulsory; you can get away without it at the price of a possible compile-time “warning” — a message that signals a possible problem but doesn’t prevent compilation. With this option, which assimilates the function name to the corresponding machine-level address, you gain the same flexibility as if you were programming in assembly language but lose the benefits of type checking.

Many-Little-Wrappers and nested classes

If a programming language does not support any of the preceding techniques but is object-oriented — with classes, inheritance, polymorphism and dynamic binding — you can use the Many-Little-Wrappers pattern studied at the beginning of this chapter.

The main disadvantage is the need to write many little classes, often with just one routine. Java, which has no agent-like mechanism and no way to pass routines as argument, mitigates this problem in part by allowing the programmer to declare a class as local to another class; this is known as a **nested class**. You can then use that class, as if it were a feature of the enclosing class, to describe objects that will only need to be created by features of the latter. This avoids polluting the global name space of the program (that is to say, the set of class names directly available to other software components); but the basic problems remain the same.

19.10 FURTHER READING

[This will include a good reference on lambda calculus accessible to novices.]

19.11 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A number of applications benefit from a mechanism for packaging a routine into an object and storing it away for later call. The corresponding language construct may be called “agent”; other common names include “delegate” (in the C# language) and “closure”.
- An agent wrapping a routine can be treated as any other objects, for example assigned to variables and passed around the program structure through feature calls. It can be called at any time through a feature applicable to all agents; this triggers a call of the associated routine, but the context of the agent’s call need not know, and usually does not know, what that routine is.
- Agents can have any number of “open operands”, corresponding to the bound variables of a lambda expression. Open operands may include some or all of the arguments, as well as the target. Closed arguments (the non-open ones) are specified in the agent’s definition; open arguments must be provided, in the form of a tuple, for each call to the agent.
- Agents can be defined on the basis of an existing routine; it suffices to specify the values of closed operands if any. To avoid defining a new routine when none is available, it is also possible to declare an agent “inline” by writing the instructions directly in the agent’s definition.
- In a programming language not supporting agents or a similar mechanism, passing functions around as data requires the use of many wrapper classes, or routines as arguments, or routine addresses. These solutions are less convenient and, in the last case, less type-safe.
- The theory of lambda calculus provides a mathematical framework for understanding agent.

- A lambda expression includes *bound variables* and a defining expression (itself possibly a lambda expression), which may involve the bound variables as well as other variables said to occur *free*. It represents a function; applying the function to arguments yields the defining expression after substitution of each argument for the corresponding bound variable. This process is known as *beta reduction*.
- The bound variables of a lambda expression are arbitrary names. They can be changed throughout the expression (including in its defining expression) as long as this doesn't create any conflicts, in particular with free variables. This process is known as *alpha conversion*.
- To *curry* a function of n arguments is to specialize it on m of its arguments ($1 \leq m < n$), leaving a function of $n - m$ arguments.

New vocabulary

Agent	Alpha-conversion	Beta-reduction
Church-Rosser property	Closed operand	Closure
First-class citizen	Inline agent	Lambda calculus
Lambda expression	Many-Little-Wrappers pattern	
Nested class	One-Song-Artist class	Open operand
Operand	Partial evaluation	Prototype (C, C++)
Substitution (of a variable in an expression)		

19-E EXERCISES

19-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

19-E.2 An integration class without agents

See the corresponding “Programming Time!”.

← Page [463](#).

19-E.3 Iterator objects

Devise an iterating mechanism that doesn't use agents but relies on a *LINEAR_ITERATOR* class describing objects able to iterate a specific operation on a linear structure such as a list.

19-E.4 Using Object Test

Rewrite the library procedure `do_all` to use Object Test rather than assignment attempt. (This does not affect “Part 2” of the body.) ← Page 470.

19-E.5 An iterator that shoots itself in the foot

(This is a masochistic type of programming exercise, asking you to violate a methodology prescription just to contemplate what mess will result.) Working with a descendant of `LINEAR` such as `LINKED_LIST`, use the procedure `do_all` with an agent argument representing a routine that — disregarding the explicit prescription in `do_all`'s header comment — changes the structure, in such a way that `do_all` crashes execution or produces an otherwise inconsistent result. With the help of the debugger if needed, analyze the exact circumstances leading to this failure.

19-E.6 Uncurrying

It was noted that currying is a one-to-one function. Write the signature and definition of the function `yrruc` that, given a one-argument function `f` whose result is a one-argument function, yields the associated two-argument function `f` such that `f = curry (f)`.

19-E.7 Beta-reduction condition

Show that the condition for beta reduction of $[\lambda x: X \mid exp] (e)$, “no free variable of e occurs bound in exp ”, is stronger than actually needed for the reduction to preserve the informal semantics of function application, and devise a less restrictive but still correct condition.

19-E.8 Alpha-conversion condition

Show that the condition for alpha-conversion of $e \triangleq \lambda x: X \mid exp$ into $\lambda y: X \mid exp [x := y]$, “ y occurs neither free nor bound in e ”, is stronger than actually needed for the reduction to preserve the informal semantics of change of variable, and devise a less restrictive but still correct condition.

20

Event-driven design

Who's in charge?

In the style of programming that we have used so far, the program defines the order of operations. It follows its own scenario, defined by control structures: sequence, conditional, loop. The external world has its say, of course, through user interaction, database access and other input, affecting the conditions that control loops and conditionals. But it's the program that decides when to test.

In this chapter we explore another kind of control structure where the program no longer specifies the sequencing of operations directly. Instead, it is organized as a set of *services* ready to be triggered in response to *events*, such as might result from a user clicking a button, a sensor detecting a temperature change, a message arriving on a communication port. At any time, the next event determines which service gets solicited. Once that service has carried out its function, the program gets back to waiting for events.

Such an **event-driven** scheme requires proper initialization: before the real action begins, there must be a setup step to associate services with events.

This architectural style — in the end another control structure to be added to our previous list — is also known as **publish-subscribe**, a metaphor emphasizing the complementary role of various elements of the software:

← Chapter 7, [Control structures](#).

- Some elements of the software, the *publishers*, trigger events.
- Some elements, the *subscribers*, register their interest in certain types of events, indicating what services they want to solicit in response.

These roles are not exclusive, as some subscribers may trigger events of their own. Note that “event” is a *software* concept: even when events originate outside the software — mouse click, sensor measurement, message arrival — they must be translated into software events for processing; and the software may trigger its own events, unrelated to any external impulse.

Event-driven programming is applicable to many different areas of programming. It has been particularly successful for Graphical User Interfaces (GUI), which we'll use as our primary example.

20.1 EVENT-DRIVEN GUI PROGRAMMING

Good old input

Before we had GUIs, programs would take their input from some sequential medium. For example a program would read a sequence of lines, processing each of them along the way:

```

from
    read_line
    count := 0
until
    exhausted
loop
    count := count + 1
    -- Store last_line at position count in Result:
    Result.put (last_line, count)
    read_line
end

```



where *read_line*, having precondition **not** *exhausted*, reads the next line of input and leaves it in *last_line*, and *exhausted* doesn't refer to the feelings of the programmer but simply states that there is no more line to be consumed.

With such a scheme **the program is in control**: it decides when it needs some input. The rest of the world — here a file, or a user typing in lines at a terminal — then has to provide that input.

Modern interfaces

Welcome to the modern world. If you write a program with a GUI, you let users choose, at each step, what they want to do, from many possibilities — including some having nothing to do with your program, since a user may choose another window, for example to answer an email.

Consider the screen on the adjacent page, from Traffic. The interface that we show to our user includes a text field and a button. There might be many more such “*controls*” (the Windows term for graphical elements, called “*widgets*” in the Unix world). We expect that the user will perform some input action, and we want to process it appropriately in our program. The action might be typing characters into the text field at the top, clicking the button, or any other, such as menu selection.

But which of these will happen first? Indeed, will any happen at all?

We don't know.

*A program
GUI*

(Screenshot to
be added.)

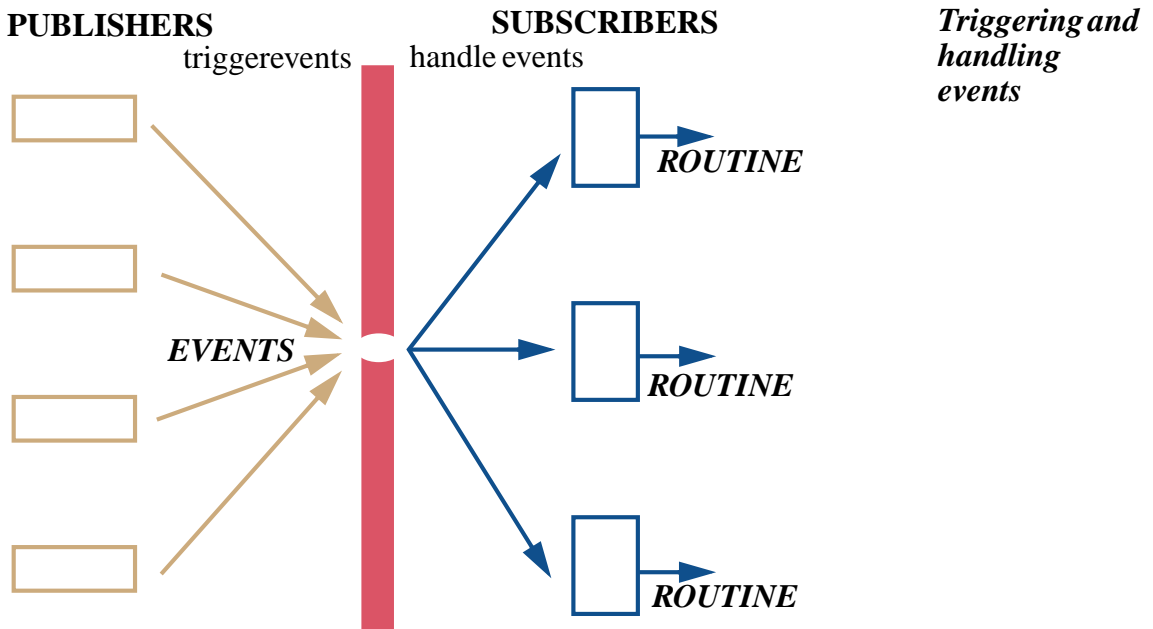
Of course we could use a big **if ... then ... elseif ... end**, or more conveniently a multi-branch listing all possibilities:

```
inspect  
  user_action  
when "Click on the Time button" then  
  "Display the time"  
when "Input in the Time field" then  
  "Update the time"  
when ... Many other branches ...  
end
```

but this suffers from all the problems we have seen with multiple-choice algorithm structures (as part of the justification for dynamic binding): it's big and complex, and highly sensitive to any change in the setup. We want a simpler and more stable architecture, which we won't have to update each time there is a new control.

Event-driven design addresses such a situation through a completely different scheme, as suggested above.

We may think of the publish-subscribe scheme as one of those quantum physics experiments (see the figure on the next page) that hurl various particles at some screen with a little hole, to find out what might show up on the other side.



This scheme works for many different application areas; GUI programming is just an example. You can find many others, in fields such as:

- Communication and networking, where a node on a network may be broadcasting messages that any other node may pick up.
- Process control. This term covers software systems associated with industrial processes, for example in factories. Such a system might have sensors monitoring temperature, pressure, humidity; any new recording, or just those exceeding some preset values, may trigger an event which some elements of the software are prepared to handle.

20.2 TERMINOLOGY

In describing event-driven programming it is important to define the concepts carefully, distinguishing in particular — as in other areas of programming — between types and instances.

Events, their types and their arguments

Definition: Event

An **event** is a signal emitted by a software element during execution.

This simple definition indicates that for our purposes (as already noted) an event is a *software* occurrence. For phenomena triggered outside of the software, such as a mouse click, we will use the term **external event**.

The distinction is important: a software system, say an application built on Traffic, does not directly react to external events. When a user clicks a mouse button, this gets processed by operating-system-level software, enabling a GUI library (such as EiffelVision) to detect the external event and trigger a software event, which other parts of the system may then handle.

This case, in which a software event is triggered as a consequence of an *external* event, is common but not the only one; in an event-driven architecture parts of the system may decide to trigger their own software-only events.

We already know the name for producers of events:

Definition: Publisher

A **publisher** is a software element that may trigger events.

To “trigger” an event is to emit the corresponding signal. This is also called “*raising*” the event or, naturally, *publishing* it.

What characterizes an event is not just that it was triggered but that it is of a certain kind, or *event type*. For example all left-button mouse clicks are of the same event kind, but a left-button click is of a different type than a key-press (keyboard input) event. This will turn out to be the principal notion:

Event types, signature

Any event belongs to a single **event type**, characterized by a **signature**.

The notion of signature was defined for routines: the signature of a procedure is a list of types, specifying that every call to the routine must provide a list of arguments of the corresponding types. (For a function, the signature also includes a result type.)

← “ANATOMY OF A ROUTINE DECLARATION”, page 201.

The reason event types have signatures is that events may have *arguments*. Other than its type, the primary property of an event is that it occurs: on 3 August 1492, Christopher Columbus set sail; five minutes ago, I clicked the left button of my mouse. But to process the event we may need some supplementary information: where did Columbus take off? What were the coordinates of my mouse cursor? Such information will appear in the event’s arguments, called that way by analogy to the information passed, through arguments, to a routine execution:

Event arguments

Every event includes a list of values, of types matching the signature, called the **arguments** of the event.

For example:

- It is not enough to know that a left-click has occurred: the software (specifically, the *subscribers* as defined next) will usually need the coordinates of the point where the click occurred. Consequently, the event type's signature is `[INTEGER, INTEGER]` (screen coordinates are generally measured as pixel distances from the origin), and any particular left click will carry a pair of integers `[z, y]`. It is also possible to have a single “mouse click” event with a third component to the signature indicating the button that was clicked. (This is the convention of the EiffelVision library, where such events have additional arguments representing such properties as the pressure applied, useful for joysticks and advanced pointing devices.)
- Although we might have an event type for each key that can be pressed on the keyboard, it's more attractive to have a single “key press” event and pass the code of the key as an argument.
- A “temperature change” event may include the old and new values.

Whenever a publisher triggers an event, it must provide a value for each argument: mouse coordinates, key code, temperatures.

Some events, such as a timeout, don't need to carry any information. They have an empty argument list; corresponding event types have an empty signature.

The notions of signature and argument make event types comparable to routines: like a procedure `f (a: TYPE1; b: TYPE2)`, an event type is characterized by a name and by a list of types. As a routine gets called, so does an event type get triggered.

The term “type” might suggest a different analogy, where event types correspond to the types of object-oriented programming (classes, possibly with generic parameters), and events to their *instances* (objects). But comparing event types to *routines* is more appropriate; then an *event* of a certain type corresponds to one specific *execution* of a routine.

In our model, then, an event is not an object — and an event type is not a class. Instead the *general notion* covering all possible event types is a class, called `EVENT_TYPE` below; and *one particular event type*, for example “left-button mouse click” (the idea of left clicks, not one particular click that happened last Monday in my office when I got fed up with the spam and clicked “OK” for “Delete all messages?”) is an object. As always when you are hesitating about whether — or, as in this case, at what level — to introduce a class, the criterion is “is this a meaningful data abstraction, with a set of well-understood operations applicable to all instances?”. Here:

- If we decided to build a class to represent a particular event type, say left click, its instances would be events of that type; but they have no useful features. True, each event has its own data (the arguments), but other than accessing such data there's no meaningful operation on an event. Its single property is that it occurred.

- In contrast, if we treat an event type as an object, the associated features are obvious and useful: trigger a particular event of this type now, with given values for the arguments; subscribe a given subscriber to this event type; unsubscribe a subscriber; find out how many events of this type have been triggered so far; find out what its subscribers are; and so on. This is the kind of rich feature set that characterizes a legitimate class.

Not treating each event as an object is also good for performance, since it is common for execution to trigger many events; each move of the mouse cursor is an event, even if it is part of a broad sweep of the cursor across the screen, so we should avoid creating all the corresponding objects — even though this does not get us out of the wood since the *arguments* of each event must still be recorded, each represented by a tuple. A good GUI library will remove the performance overhead by recognizing a sequence of contiguous moves in close succession and allocating just one tuple instead of dozens or hundreds.

← Class *EVENT_TYPE*
in the final design:
[“USING AGENTS:
THE EVENT
LIBRARY”](#), 20.5, page
523.

Events of a given event type will be of interest to certain parts of the software:

Definitions: Subscriber, Register, Subscribe

A **subscriber** is a software element that **registers** (or “**subscribes**”) to be notified of events of a certain event type, so that it can execute specified actions in response.

A subscriber registers for an event type, not a particular event. Although registration (and deregistration) may occur at any time, it is common to have an initialization phase that puts the basic subscriptions in place.

The same software element may, as noted, act as both a publisher and a subscriber; in particular it is a common scheme for a subscriber to react to an event by triggering another event.

Registering, for a subscriber, means specifying a certain **action** for execution in response to any event of the specified type. There must be a way for the action to obtain the values of the event’s arguments. The obvious way to achieve such registration is to specify a **procedure**, whose signature matches the event type’s signature.

We also have terms describing how a subscriber reacts to an event:

Definitions: Handle, catch

When a subscriber gets notified of an event to whose type it has subscribed, it **handles** (or “**catches**”) the event by executing the registered action.

We now have the full picture of how an event-driven design works:

- 1 • Some elements, *publishers*, make known to the rest of the system what *event types* they may trigger.
- 2 • Some elements, *subscribers*, are interested in *handling* specific event types. They *register* the corresponding actions.
- 3 • At any time, a publisher can *trigger* an event. This will cause execution of actions registered by subscribers for the event’s type.

In the GUI example:

- 1 • A publisher is some element of the software that tracks input devices such as the mouse and the keyboard, and triggers events under specified circumstances, for example mouse click or key press. You usually don't have to write such software yourself; rather, you rely on a **GUI library** — such as EiffelVision for Eiffel, Swing for Java and Windows Forms for Microsoft's .NET — that takes care of triggering the right events.
- 2 • A subscriber is any element that needs to handle such GUI events; they register the routines they want to execute in response. For example we may register, for occurrences of the mouse click event type on the Time button, a routine that displays the time.
- 3 • If, during execution, a user clicks the Time button, this will cause execution of the routine — or routines — registered for this kind of event.

An important property of this scheme, illustrated by the separation between the two sides in our earlier [figure](#), is that subscribers and publishers should not have to know about each other.

Keeping the distinction clear

We have made a careful distinction between events and event types. You might think it obvious, but in fact — this is a warning, to help you understand the literature if you start using various event-driven programming mechanisms — many descriptions confuse the two; this can make simple things sound tricky.

Below is an excerpt from the presentation of event handling in .NET, a virtual-machine-based framework developed by Microsoft, whose concepts are also reflected in the C# and Visual Basic .NET languages. The excerpt comes from introductory paragraphs in the online [documentation](#).

Events Overview

Events have the following properties:

- 1 • The publisher determines when an **event** is raised; the subscribers determine what action is taken in response to the **event**.
- 2 • An **event** can have multiple subscribers. A subscriber can handle multiple **events** from multiple publishers.
- 3 • **Events** that have no subscribers are never called.
- 4 • **Events** are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- 5 • When an **event** has multiple subscribers, the *event* handlers are invoked synchronously when an **event** is raised. To invoke **events** asynchronously, see [another section].
- 6 • **Events** can be used to synchronize threads.
- 7 • In the .NET Framework class library, **events** are based on the **EventHandler** delegate and the **EventArgs** base class.

← [“Triggering and handling events”](#), page 504.

From msdn2.microsoft.com/en-us/library/awbfidfj.aspx, as of August 2006. Numbers, italics and colors added.

I have highlighted in green those occurrences of “event” where the authors really mean event, and in yellow those for which they mean event type (a term that does occur in the .NET documentation, but rarely). Where the word is in *italics*, it covers both. All this is my interpretation, but I think that on the basis of the previous discussions you will agree. In particular:

- It is not possible (points [1](#), [5](#)) to subscribe to an event; for one thing, the event does not exist until it has been raised, and when it has been raised that’s too late! (Nice idea, though: wouldn’t you like to subscribe retroactively to the event “X’s shares rose by at least 20% today” for every company X listed on the NYSE?) A subscriber may subscribe to an event *type* — that is to say, declare its intention of being notified of any *event* of that type raised during execution.
- Point [7](#) talks about properties of *classes* describing event *types*, as indeed in .NET every event type must be declared as a class. Such a class must inherit from the “delegate” class `EventHandler` (.NET delegate classes provide a complicated kind of agent mechanism) and use another class `EventArgs` describing the notion of event arguments.
- Point [3](#) sounds mysterious until you realize that it means: “If an *event type* has no subscriber, triggering an *event* of that type at run time has no effect.” So all this clause describes is a performance optimization: by detecting that an event type has no subscriber, the .NET event mechanism can remove the overhead of raising the corresponding events, which in .NET implies creating an object for each. (The mystery is compounded by the use of “call” for what the rest of the .NET documentation calls “raising” an event.)
 “Event handler” ([5](#)) may mean either the subscriber registered to handle a given event *type*, or the subscriber that at run time handles a particular *event* of that type.

The possibility of confusion is particularly vivid in two places:

- “A *subscriber can handle multiple events from multiple publishers*” (point [2](#)): this might seem to suggest some concurrent computation scheme whereby a subscriber catches events from various places at once. In fact the observation is simply that you can register for several event types, and of course various publishers may trigger events of those types.
- Point [5](#) states that when “*an event*” has multiple subscribers, each will handle it synchronously (meaning right away, blocking further processing) when “*an event*” is raised. Read literally, this would suggest that two “events” are involved! That’s not the idea: the sentence is simply trying to say that when multiple subscribers have registered for a certain event *type*, they handle the corresponding *events* synchronously. In the same breath it uses a single word with two different meanings.

So when you read about event-driven schemes please make sure to ask yourself whether people are talking about events or event types (and, since this is the time for exhortations of good conduct, make sure that your own technical documentation uses precise terminology, and defines it clearly).

Contexts

A subscriber that registers says: “for events of *this type*, execute *that action*”. In practice it may be useful, especially for GUI applications, to provide one more piece of information: “for events of this type occurring *in this context*, execute that action”. For example:

- “If the user clicks the left button *on the EXIT icon*, terminate execution”.
- “If the mouse enters *this window*, change the border color to red”.
- “If this sensor reports a temperature *above 25° C*, ring the alarm”.

In the first case the “context” is an icon and the event type is “mouse click”; in the second, they are a window and “mouse enter”; in the third, a temperature sensor and a measurement report.

For GUI programming, a context is usually just a user interface element. As the last example indicates, the notion is more general; a context can in fact any boolean-valued condition. This covers the GUI example as a special case, taking as boolean condition a property such as “the cursor is on this icon” or “the cursor has entered that window”.

We can use a general definition:

Definition: Context

In event-driven design, a **context** is a boolean expression specified by a subscriber at *registration* time, but evaluated at *triggering* time, such that the registered action will only be executed if it the value is **True**.

Even though that wasn’t event-driven programming, we had a taste of the notion of context when encountering iterators such as *do_if* which performs an action on all the items of a structure that satisfy a certain condition; this is similar to how a context enables a subscriber to state that it is interested in events of a certain type but only if a certain condition holds at triggering time.

We could do without the notion of context by including the associated condition in the registered action itself, which we could write, for example

```
if “The cursor is on the Exit icon” then
    “Normal code for the action”
end
```

but it is more convenient to separate the condition by specifying it, along with the event type and the action, at the time of registration.

20.3 PUBLISH-SUBSCRIBE REQUIREMENTS

With the concepts in place, we will now look for a general solution to the problem of devising an event-driven architecture. We start with the constraints that any good solution must satisfy.

Publishers and subscribers

In devising a software architecture supporting the publish-subscribe paradigm we should consider the following requirements:

- *Publishers must not need to know who the subscribers are:* they trigger events, but do not know who is going to process those events. This is typically the case if the publisher is a GUI library: the routines of the library know how to detect a user event such as a mouse click, but they should not have to know about any particular application that reacts to these events, or how it reacts. To an application, a button click may signal a request to start a compilation, run the payroll, shut down the factory or launch the rocket. To the GUI library, a click is just a click.
- *Any event triggered by one publisher may be consumed by several subscribers.* An event such as the change of a temperature in a factory control system may have to be reflected in many different places that “observe” it, for example an alphanumeric temperature display, a graphical display, a database that records all value changes, and a security system that triggers certain actions if the value is beyond preset bounds.
- *The subscribers should not need to know about the publishers.* This is a more advanced requirement, but often desirable too: subscribers know about events to which they subscribe, but do not have to know where these events come from. Remember that one of the fundamental aims of an event-driven architecture is to provide a flexible architecture where we can plug in various publishers and various subscribers, possibly written by different people at different times, without any dependency.
- *You may wish to let subscribers register and deregister while the application is running.* The usual scheme is that registration occurs during initialization, to set things up before “real” execution starts; but this is not an obligation, and the extra flexibility may be useful.
- *It should be possible to connect publishers and subscribers at minimal work.* The actions to be subscribed often come from an existing application, to which you want to add a GUI or other interface. To connect the two sides you’ll have to add some program text, often called “**glue code**”; the less of it the better.
- *It should be possible to make events dependent or not on a context.* We have seen the usefulness of binding events to contexts, but the solution should also provide the ability — without having to define an artificial context — just to subscribe to an event regardless of where it happens.

The model and the view

In the particular case of GUI programming we don't just need to separate subscribers from publishers but also to keep the *model* separate from the *view*. These are two complementary aspects of an application:

Definitions: model, view of a software system

The **model** (also called *business model*) is the part of a software system that handles data representing information from the application domain.

A **view** is a presentation of part of that information, in the system's interaction with the outside: human users, material devices, other software.

← [“Definitions: Data, information”, page 10.](#)

“*Application domain*” as used in this definition is also a common phrase, denoting the technical area in which or for which the software operates. For a payroll processing program the application domain is human resources of companies; for a system that handles bank accounts it's finance; for a text preparation program like Microsoft Word or StarOffice it is text processing; for flight control software the application domain is air traffic control.

While the application domain need not have anything to do with software, the “*model*” is a part of the software: the part that deals with that application domain. For the payroll processing program it's the part of the software that processes information on employee salaries, hours worked and deductions, computes salaries, updates the database. For the flight control system it's the part that determines airplane itineraries, takeoff times, authorizations and so on. One could say that the model is the part of the software that does the “real job” at hand, independently of other aspects such as interaction with users of the software.

“Business model” is more precise but we usually just say “model” because the word “business” might be misinterpreted as restricting us to business-oriented application domains (company management, finance etc.) at the expense of engineering domains such as text processing and flight control.

A “*view*” is a presentation of the information, typically for input or output. A GUI is a view: for example the flight control system has a user interface allowing controllers to follow plane trajectories and enter their own information and commands.

Usually a program covers just one — possibly broad — application domain, but it may have more than one view, hence “the model” and “a view” in the above definition. It is then usually desirable to assign the two aspects to two different parts of a system's architecture. In a naïve design for a small program you might not pay much attention to this issue. But in a significant system you should, if only because it is important to plan for *several views*. Typically you might need some or possibly all of:

- A GUI view.
- A Web view, allowing use of the program through a Web browser.
- A purely textual (non-graphical) interface, for situations in which graphics support is not available.
- A “*batch*” interface where the system takes its input from a prepared scenario and produces its output globally. This notion of scenario is particularly useful for testing interactive systems: it is very difficult to test them interactively, as this would require people spending endless sessions with the system to try many different combinations. Instead you may prepare a large collection of scenarios (which may include some automatically recorded from previous sessions with human users) and run them without human interaction.
- An API, permitting interaction with other programs running locally.
- A *Web service* view, similar to an API but intended for programs accessing the facilities with HTTP (the Web protocol) across the Internet.

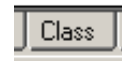
Often one view is enough at the beginning; that’s why it is a common design mistake to make the model and the view intricately connected. Then when you need to introduce other views you may be forced to perform extensive redesign. To avoid this you should practice model-view separation as a general principle, right from the start of a design:

Touch of Methodology: **Model-View Separation Principle**

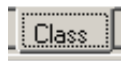
In designing the architecture of a software system, keep the coupling between model elements and view elements to a minimum.

If we use an event-driven model this rule goes well with a clear separation of publishers and subscribers. Both the subscribers and the publishers will interact with the view, but in a decoupled way:

- Publishers trigger events which may immediately update the view, typically in minor ways; for example the cursor may change shape when it enters a certain window, and a button usually changes its aspect when it has been pressed (like the Class button on the right).
- Subscribers catch events (of event types to which they are subscribed), and process them. This processing may update the view.



Not pressed



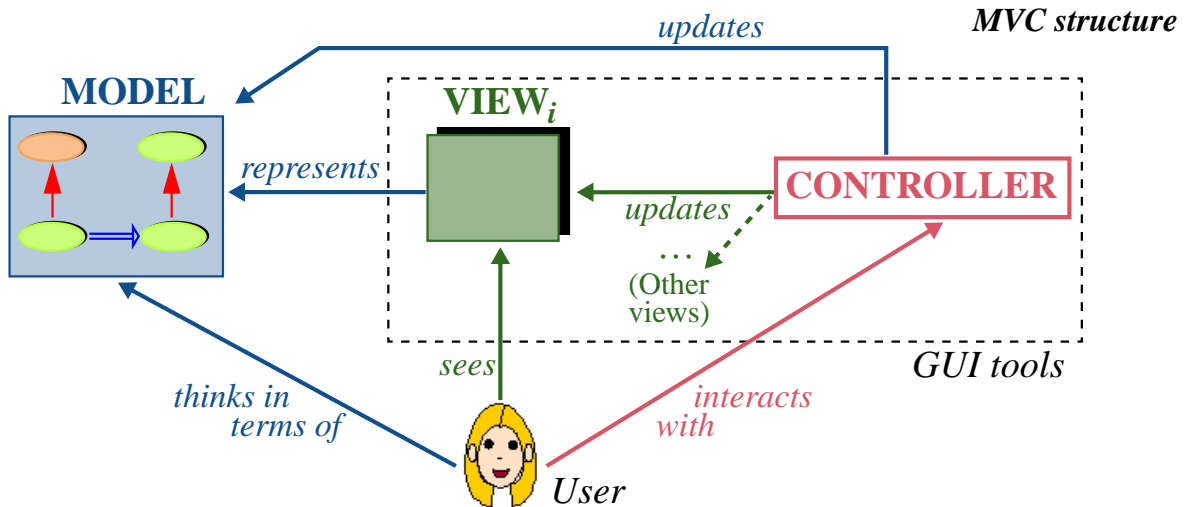
Pressed

Note that the publisher-subscriber and model-view divisions are orthogonal: both publishers and subscribers may need to interact with the model as well as with the view, as we can see in the example of a text processing system:

- The reason for a publisher to trigger an event may come from the view — a user moves the mouse or clicks a button — or from the model, as when the spell checker detects a misspelled word and highlights it visually.
- The processing of an event by a subscriber will often cause modifications both to the model and to the view. For example if the user has selected a certain text and then presses the Delete key, the effect must be both to remove the selected part from the representation of the text kept internally by the system (model) and to update the display so that it no longer shows that part (view). *Selecting text for deletion* (Illustration to be added)

Model-View-Controller

A scheme often recommended for GUI design is “Model-View-Controller” or MVC. The third element, Controller, directs the execution of an interactive session. Each of the three parts communicates with the other two:



The role of the controller is to provide further separation between the model and the views. (Remember that there may be more than one view, hence “VIEW_i” in the figure.) The controller handles user actions, which may lead to updates of the view, the model, or both.

As before, a view provides a visual representation of the model or part of it.

Users are assumed to understand the model: using a text processing system, I should know about fonts, sections and paragraphs; playing a video game, I should have a feel for rockets and spaceships. A good system enables its users to *think* in terms of the model: even though what I see on the screen is no more than a few pixels making up some circular shape, I think of it as a flying vessel. The controller enables me to act on these views, for example by rolling my mouse wheel to make the vessel fly faster; it will then update both

the model, by calling features of the corresponding objects to change their attributes (speed, position), and the view, by reflecting the effect of these changes in the visual representation.

MVC provides a useful paradigm and has had considerable influence on the spread of graphical interactive application over the past decades. We will see at the end of this chapter that by taking the notion of event-driven design to its full consequences we can get the benefits of MVC but with a simpler architecture, bypassing some of the relations that populate the last figure.

A side comment on this figure, serving as general advice. Far too often in presentations of software concepts — and elsewhere too, but it’s particularly bad in information technology — you will find impressive-looking diagrams with lots of boxes connected by lots of arrows, but little reassurance of what they mean. It is easy to draw arrows, but better to state what they stand for (their “semantics”); this is the role of the arrow labels on the last figure, such as “*represents*”, “*updates*” etc. (Even the unlabeled arrows in the “model” part on the left reflect our standard conventions, suggesting client and inheritance relations between classes of the model.) Please remember this when you propose figures, or see other people’s figures: a picture will *not* be worth any number of words if it’s just splashes of fancy-looking color. Don’t succumb to the lure of senseless graphics; assign precise semantics to each graphical symbol you use, and document it.

20.4 THE OBSERVER PATTERN

Before we review what will be the definitive scheme for event-driven design (at least for the kind of examples discussed in this chapter), let’s explore a well-known *design pattern*, “Observer”, which also addresses the problem.

About patterns

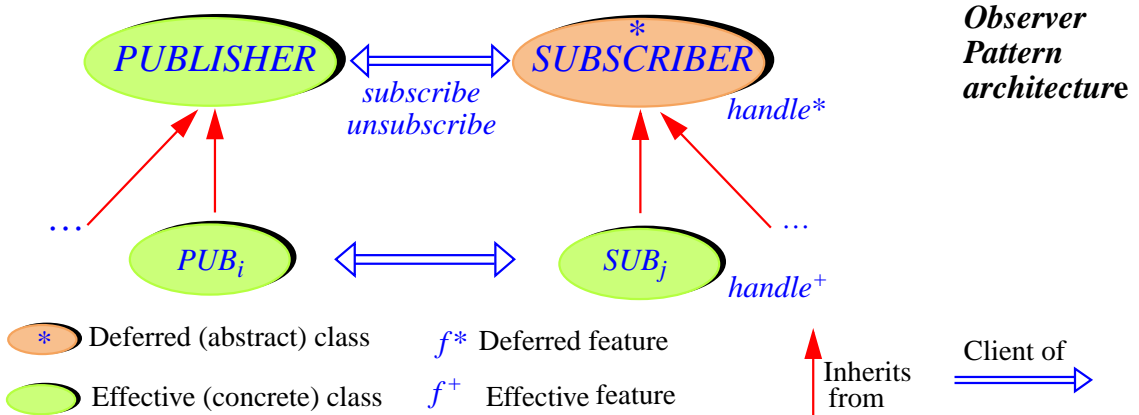
A design pattern is a standardized architecture addressing a certain class of problems. Such an architecture is defined by typical classes that must be part of the solution, their role, their relations — how they inherit from and are clients of each other —, and instructions for customizing them as the problem varies. Design patterns emerged in the mid-nineties as a way to record and catalog design solutions that good programmers had devised over the years, often reinventing them independently: “best practices”, as they are sometimes known. A couple dozen of these patterns, Observer among them, are widely documented and taught; hundreds more have been described or proposed.

Observer basics

As a general solution for event-driven design, Observer is actually not very good; we’ll analyze its limitations. But you should know about it anyway for several reasons: it’s a kind of classic; it elegantly takes advantage of O-O techniques such as polymorphism and dynamic binding; it may be your best

bet if you are using a language that doesn't support such notions as agents, genericity and tuples; and it provides a good basis for moving on to the more satisfactory solution studied next.

The following figure illustrates the typical organization of an Observer architecture. *PUBLISHER* and *SUBSCRIBER* are two general-purpose classes, not specifically dependent on your application; *PUB_i* and *SUB_j* stand for typical publisher and subscriber classes in your application.



To discuss publish-subscribe schemes we may say that the subscribers “observe” the publishers, remaining on the alert for any messages from them, and that the publishers are the “subjects” of this observation. This explains why some of the literature uses *observer* and *subject* instead of “subscriber” and “publisher”. Either terminology is fine; the underlying ideas are the same. You will similarly encounter, in the patterns literature, other names for the key features: “attach” for *subscribe*, “detach” for *unsubscribe*, “notify” for *publish*, “update” for *handle*.

Although both *PUBLISHER* and *SUBSCRIBER* are intended to serve as ancestors to classes doing the actual job of publishing and handling events, only *SUBSCRIBER* must be deferred; its procedure *handle* will define, as effected in each concrete subscriber class *SUB_j*, how the given kind of subscriber handles events. *PUBLISHER* does not have such a deferred feature, although we could still make it deferred too to preclude direct instantiation.

The publisher side

Class *PUBLISHER* describes the properties of a typical publisher in charge of an event type — that is to say, of triggering events of that type, through procedure *publish*. The principal data structure is a list of *subscribers* to that event type:

```

note
  what: ["Objects that can publish events, all of the same type,
         monitored by subscribers"]
class
  PUBLISHER
feature {SUBSCRIBER} -- Status report
  subscribed (s: SUBSCRIBER): BOOLEAN
    -- Is s subscribed to this publisher?
  do
    Result := subscribers.has (s)
  ensure
    present: has (s)
  end
feature {SUBSCRIBER} -- Element change
  subscribe (s: SUBSCRIBER)
    -- Make s a subscriber of this publisher.
  do
    subscribers.extend (s)
  ensure
    present: subscribed (s)
  end
  unsubscribe (s: SUBSCRIBER)
    -- Make s a subscriber or this publisher.
  do
    subscribers.remove_all_occurrences (s)
  ensure
    absent: not subscribed (s)
  end
  publish (args: LIST [ANY]) -- Argument Scheme 1
    -- Publish event to subscribers.
  do
    ... See below ...
  end
feature {NONE} -- Implementation
  subscribers: LINKED_LIST [SUBSCRIBER]
    -- Subscribers subscribed to this publisher's event.
end

```

See below about *publish*, the type of its argument, and its "Argument Scheme".

The implementation is somewhat primitive since it doesn't prohibit calling *attach* twice for the same subscriber; then (see *publish* below) such a subscriber would execute the subscribed action twice when the event is published — most likely not the desired effect. To avoid this we could enclose the implementation of *attach* in **if not subscribed (s) then ... end**, but then the linked list is no

longer an efficient implementation since *has* requires a traversal. While not critical to the present discussion, this matter must be addressed properly for any actual use of the pattern; it's the subject of an exercise.

Apart from *subscribers*, meant for internal purposes only and hence secret (exported to *NONE*), the features are relevant subscriber objects but not to any others; they are hence exported to *SUBSCRIBER*, which means, as you will remember, that they are also exported to the descendants of this class, which will indeed need to *subscribe* and *unsubscribe* the corresponding objects. As a general rule, it is a good idea to export features selectively when they are only intended for specific classes and their descendants. Better err on the side of restrictiveness here to avoid mistakes caused by classes calling features that are none of their business; it's easy later on to ease the restrictions if you find some new classes need the feature.

Procedure *publish* will notify all subscribers that an event (of the event type for which the publisher is responsible) has occurred. It will be easier to write it after devising the class representing a typical subscriber.

→ "[Efficient Observer](#)", 20-E.2, page 532.

The subscriber side

```

note
  what: "Object that can register to handle events of a given type"
deferred class
  SUBSCRIBER
feature -- Element change
  subscribe (p: PUBLISHER)
    -- Subscribe to p.
    do
      p.subscribe (Current)
    ensure
      present: p.subscribed (Current)
    end
  unsubscribe (p: PUBLISHER)
    -- Ensure that this subscriber is not subscribed to p.
    do
      p.unsubscribe (Current)
    ensure
      absent: not p.subscribed (Current)
    end
feature {NONE} -- Basic operations
  handle (args: LIST [ANY]) -- Argument Scheme 1
    -- React to publication of one event of subscribed type
  deferred
  end
end

```

See below about the "Argument Scheme" and the type of *args*.

This class is deferred: any class of an application can, if its instances may need to act as subscribers, inherit from *SUBSCRIBER*. We'll call such descendants “subscriber classes” and their instances “subscribers”.

To subscribe to a given event type, through the corresponding publisher *p*, a subscriber simply executes *subscribe (p)*. Note how this procedure (and, similarly, *unsubscribe*) uses the corresponding mechanism in *PUBLISHER* to subscribe the current object. That was one of the reasons for exporting the *PUBLISHER* features selectively: it would make no sense for potential subscribers to use *subscribe* from *PUBLISHER* directly, since this is only useful if you also provide the corresponding *handle* mechanism; the feature of general interest is the one from *SUBSCRIBER*. (This also justifies using the same names for the features in the two classes, which keeps the terminology simple and causes no confusion since only the *SUBSCRIBER* features need be widely known.)

Each observer class will provide its own version of *handle*, describing how it handles an event. The not so pleasant part is accessing arguments if any; that's because we tried to make *PUBLISHER* and *SUBSCRIBER* general, and so had to declare *args*, representing the event arguments in both *publish* and *handle* in these respective classes, a completely general type, *LIST [ANY]*; but then *handle* has to force the right type and number of arguments if it can. For example, to process a mouse click event with the *[x, y]* coordinates as argument — by calling some *operation* which takes these values as its own routine arguments — we may use

```

handle (args: LIST [ANY])                                     -- Argument Scheme 1
  -- React to publication of mouse click event by performing
  -- operation on the cursor coordinates.
do
  if args.count >= 2 and then
    ({x: REAL} (args.item (1)) and {y: REAL} (args.item (2)))
  then
    operation (x, y)
  else
    -- Do nothing, or report error
  end
end
end

```

The Object Tests make sure that the first and second argument are *REALs*, and binds them to *x* and *y* within the **then** clause. The only way to avoid this awkward run-time testing of argument types would be to specialize *PUBLISHER* and *SUBSCRIBER* by declaring the exact arguments to *publish* and *subscribe*, for example

```
publish (x, y: REAL)
```

```
-- Argument Scheme 2
```

and similarly for *handle* in *SUBSCRIBER*. This loses the generality of the scheme since you can't use the same *PUBLISHER* and *SUBSCRIBER* for event types of different signatures. Although it's partly a matter of taste, I would actually recommend this “[Argument Scheme 2](#)” if you need to use the Observer pattern, because it will detect type errors — a publisher passing the wrong types of arguments to an event — at compile time, where they belong. With *handle* as written above you'll only find them at run time, through the tests on the length and element types or *args*; that's too late to do anything serious about the issue, as reflected by the rather lame “[Do nothing, or report error](#)”: doing nothing means ignoring an event (is that what we want, even if the event is somehow deficient since it doesn't provide the right arguments?); and if we report an error, report it to whom? The message should be for the developers — you! — but it's the poor end user who will get it.

It was noted in the discussion of object test that this mechanism should generally be reserved for objects coming from the outside, not those under the program's direct control, for which the designer is in charge of guaranteeing the right types statically. Here the publishing and handling of arguments belong to the same program; using object test just doesn't sound right.

It is actually possible to obtain a type-safe solution by making classes *PUBLISHER* and *SUBSCRIBER* generic; the generic parameter is a tuple type representing the signature of the event type (that is to say, the sequence of argument types). That solution will appear in the final publish-subscribe architecture below (“Event Library”). We won't develop it further for the Observer pattern because it relies on mechanisms — tuple types, constrained genericity — that are not all available in other languages: if you are programming in Eiffel, which has them, you should use that final architecture (relying on agents), which is much better than an Observer pattern anyway and is available through a prebuilt library. It is a good [exercise](#), however, to see how to improve Observer through these ideas; try it now on the basis of the hints just given, or wait until you have seen the solution below.

→ [“Type-safe Observer”, 20-E.3, page 533.](#)

Publishing an event

The only missing part of the Observer pattern’s implementation is the body of the *publish* procedure in *PUBLISHER*, although I hope you have already composed it in your mind. It’s where the pattern gets really elegant:

```
publish (args: ... Argument Scheme 1 or 2, see above discussion ...)
  -- Publish event to subscribers.
  do
    -- Ask every subscriber in turn to handle the message:
    from subscribers.start until subscribers.after loop
      subscribers.item.handle (args)
    subscribers.forth
  end
end
```

← To be inserted in class *PUBLISHER*, page 517.

(With “Argument Scheme 2” the arguments passed to *handle* will be more specific, for example *x* and *y* for a mouse click.) The highlighted instruction takes advantage of polymorphism and dynamic binding: *subscribers* is a polymorphic list; each item in the list may be of a different *SUBSCRIBER* type, characterized by a specific version of *handle*; dynamic binding ensures that the right version is called in each case.

Assessing the Observer pattern

The Observer pattern is widely used and known, and is an interesting application of object-oriented techniques. As a general solution to the publish-subscribe problem it suffers from a number of limitations:

- The business of arguments, as discussed, is unpleasant, causing a dilemma between two equally unattractive schemes: awkward, type unsafe run-time testing of arguments, and specific, quasi-identical *PUBLISHER* and *SUBSCRIBER* classes for each event type signature.
- Subscribers directly subscribe to publishers. This causes undesirable coupling between the two sides. Subscribers shouldn’t need to know which part of an application or library triggers certain events. What we are really missing here is an intermediary — a broker, if you like — between the two sides. The more fundamental reason is that the design has missed an important abstraction: the notion of event type, merged here with the notion of publisher.

- A subscriber may register with only one publisher; with that publisher, it can register only one action, as represented by *handle*. As a result, it may subscribe to only one type of event. This is severely restrictive. An application component should be able to register various operations with various publishers. It is possible to address this problem by adding to *publish* and *handle* an argument representing the publisher, letting subscribers discriminate between publishers; but this solution is detrimental to modular design since the handling procedures will now need to know about all events of interest.
- Because publisher and subscriber classes must inherit from *PUBLISHER* and *SUBSCRIBER*, it is not easy to connect an existing model (in the sense defined above) to a new view, for example a GUI, without adding significant glue code. In particular, you can't directly reuse an existing procedure from the model as the action to be registered by a subscriber: you have to fill in the implementation of *handle* so that it will call that procedure, with the arguments passed by the publisher.

The above classes already contains a number of improvements over the implementations of the Observer pattern which you'll find in much of the literature and which actually cause further problems. For example the standard presentation binds a subscriber to a publisher at *creation* time, using the publisher as an element to the observer's creation procedure. The above implementation provides instead a *subscribe* procedure in *OBSERVER*, to bind the observer to a specific publisher when desired; so at least you can later unsubscribe, and re-subscribe to a different publisher.

It's also worth pointing out that *PUBLISHER* and *SUBSCRIBERS*, intended to be inherited by publisher and subscriber classes, both need effective features, such as *publish* with its fundamental algorithm and *subscribe*. This property causes a serious problem in languages such as Java and C# which do not support multiple inheritance, since it prevents publisher and subscriber classes from having other parents as may be required by their role in the application. The only solution is to write special classes and make them clients the "real" publishers and subscribers — more glue code.

All these problems have not prevented designers from using Observer successfully, but they have two serious consequences. First, the resulting solutions lack flexibility; they may cause unnecessary work, for example writing of glue code, and unnecessary coupling between elements of the software, which is always bad for the long-term evolution of the system. Second, they are not *reusable*: each programmer must rebuild the pattern for every system that needs it, adapting it to the system's particular needs.

The preceding assessment of "Observer" is an example of how one may analyze a proposed *software architecture*. Use it as a guide when presented with possible design alternatives. The criteria are always the same: reliability (decreasing the likelihood of bugs), reusability (minimizing the amount of work to integrate the solution into a new program), extensibility (minimizing adaptation effort when the problem varies), and simplicity.

→ See "[Touch of Methodology: Assessing software architectures](#)", page 530.

20.5 USING AGENTS: THE EVENT LIBRARY

We are now going to see how, by giving the notion of event type its full role, we can obtain a solution that removes all these limitations. It is not only more flexible than what we have seen so far, and fully reusable (through a library class that you can use on the sole basis of its API); it's also much simpler. The key boost comes from the agent mechanism.

We focus on the essential data abstraction resulting from the discussion at the beginning of this chapter: event type. We won't have *PUBLISHER* or *SUBSCRIBER* classes any more, but just one class — yes, a single class solves the entire problem — called *EVENT_TYPE*.

Fundamentally, two features characterize an event type:

- **Subscribing:** a subscriber object can register its interest in the event type by subscribing a specified action, to be represented by an agent.
- **Publishing:** triggering an event.

We can benefit from language mechanisms to take care of the most delicate problems identified above:

- Each event type has its own signature. We can define the signature as a tuple type, and use it as generic parameter to *EVENT_TYPE*.
- Each subscription should subscribe a specific action. We simply pass this action as an agent. This means in particular that we'll be able to reuse an existing feature from the business model.

These observations are enough to give us the interface of the class:

note

what: "*Event types, allowing publishing and subscribing*"

class *EVENT_TYPE* [*ARGUMENTS* → *TUPLE*] **feature**

publish (*args*: *ARGUMENTS*)

-- Trigger an event of this type.

subscribe (*action*: *PROCEDURE* [*ANY*, *ARGUMENTS*])

-- Register *action* to be executed for events of this type.

unsubscribe (*action*: *PROCEDURE* [*ANY*, *ARGUMENTS*])

-- De-register *action* for events of this type.

end

*Class interface only.
The implementations of
publish and subscribe
appear below.*

If you are an application developer who needs to integrate a publish-subscribe scheme in a system, the above interface — for the class as available in the Event Library — is all you need to know. Of course we'll look at the internals too, as I am sure you'll want to see them. (It will actually be more fun if you try to devise them yourself first.) But for the moment let's look at how a typical client programmer, knowing only the above, will achieve publish-subscribe.

The first step is to define an event type. This simply means providing an instance of the above library class, with the appropriate actual generic parameters. For example, you can define

```

left_click: EVENT_TYPE [ TUPLE [x: REAL; y: REAL] ]           [1]
  -- Event type representing left-button click events
  once
  create Result
  end

```

This defines an object that represents the event type. Remember, we don't need an object per event (as was the case for example in the .NET scheme); that would be a waste of space; we only need an object per event *type*, such as left-click here. Because this object will be shared by several parts of the software — publishers and subscribers for the event type — we create just one object, by using a **once** function. One of the advantages is that you don't need to worry about when to create the object; it will be created on first use.

We'll see in just a moment where the declaration of the event type, here *left_click*, should be; until then let's assume that subscriber and publisher classes both have access to it.

To trigger an event, a publisher — for example a GUI library element that detects a mouse click — simply calls *publish* on this event type object, with the appropriate argument tuple; in our example:

```

left_click.publish ([your_x, your_y])

```

On the subscriber side things are equally simple; to subscribe an action represented by a procedure *p* (*x*, *y*: REAL), it suffices to use

```

left_click.subscribe (agent p)                               [2]

```

This scheme has considerable flexibility, achieved in part through the answer to the pending question of where to declare the event type. If you want to have a single event type published to all potential subscribers, just make it available to both publisher and subscriber classes by putting its declaration [1] in a class

to which they all have access, for example by inheriting from it (a “facilities class”). On the other hand, since the event type is just an ordinary object, and the corresponding features such as *left_click* ordinary features that may belong to any class, you can declare *left_click* as a feature of one of the classes representing graphical widgets, such as *BUTTON*. Then a typical subscription call becomes

<i>your_button</i> . <i>left_click</i> . <i>subscribe</i> (agent <i>p</i>)	[3]
---	-----

if, as is usually the case, a subscriber only wants to monitor — “observe”, in Observer pattern terminology — mouse events from one particular button of the GUI. This directly implements the notion of **context** introduced earlier; here the context is simply the button.

More generally, if the context is relevant — that is to say, subscribers don’t just subscribe to an event type as in [2], but to events occurring in a context, as in [3] — then the appropriate architectural decision for event types is to declare them as feature of the appropriate context classes. For example the declaration of *left_click* [1] becomes part of a class *BUTTON*. It remains a **once** function, since the event type is common to all buttons of that kind; the event type object will be created on the first *subscribe* or *publish* call (whichever comes first). Note that if left-click is relevant for several kinds of widget — buttons, windows, menu entries ... — then each of the corresponding classes will have an attribute such as *left_click*, of the same type. This is the technique used by the EiffelVision library.

So we get the appropriate flexibility, and can tick off the last item on our list of requirements for a publish-subscribe architecture:

- For events that are relevant independently of any context information, declare the event type in a generally accessible class. (In that case the particular generic derivation of *EVENT_TYPE* describing the event type will have at most one run-time instance, shared by all that need it.)
- If a context is needed, declare the event type as a feature of a class representing contexts; it will be accessible at run time as a property of a specific context object. (In that case there can be as many instances of the *EVENT_TYPE* type as there are context types for which it is relevant.)

← “*PUBLISH-SUBSCRIBE REQUIREMENTS*”, 20.3, page 511.

Now for the internal picture. It remains to see the implementation of *EVENT_TYPE*. It is similar to the above implementation of a *PUBLISHER*. A secret feature *subscribers* keeps the list of subscribers. Its signature is now

<i>subscribers</i> : <i>LINKED_LIST</i> [<i>PROCEDURE</i> [<i>ANY</i> , <i>ARGUMENTS</i>]]

(where, as before, *LINKED_LIST* is a naïve structure but sufficient for this discussion; for a better one look up the actual class text of *EVENT_TYPE* in the Event Library, or do the [exercise](#)). The items we store in the list are no longer “subscribers”, a notion that doesn’t play a particular role any more, but simply agents, with a precise types: they must represent procedures that take arguments of the tuple type *ARGUMENTS*, as defined for the class. This considerably improves the type safety of the solution over what we saw previously: mismatches will be caught at compile time as bad arguments to *subscribe*.

→ “[Efficient Observer](#)”, 20-E.2, [page 532](#).

For *subscribe* it suffices (in the “naïve” implementation) to perform

```
subscribe (action: PROCEDURE [ANY, ARGUMENTS])
  -- Register action to be executed for events of this type.
do
  subscribers.extend (action)
ensure
  present: subscribers.has (action)
end
```

The use of *ARGUMENTS* as the second generic parameter for the *PROCEDURE* type of *action* ensures compile-time rejection of procedures that do not take arguments of a matching type.

To publish an event we traverse the list and call the corresponding agents. The code is in fact the same as before, although *args* is now of a more appropriate type, *ARGUMENTS*:

```
publish (args: ARGUMENTS)
  -- Publish event to subscribers.
do
  -- Trigger an event of this type.
  from subscribers.start until subscribers.after loop
    subscribers.item.call (args)
    subscribers.forth
  end
end
```

← To be inserted in class *PUBLISHER*, [page 517](#).

Any argument to the agent feature *call* must be a tuple; this is indeed the case since *ARGUMENTS* is constrained to be a tuple type.

The solution just describes is at the heart of the “Event Library”, and also of the EiffelVision GUI library; it is widely used for graphical applications, some small and some very complex.

20.6 SOFTWARE ARCHITECTURE LESSONS

The designs reviewed in this chapter yield a few general observations about software architecture.

Choosing the right abstractions

The most important issue in software design, at least with an object-oriented approach, is to identify the right classes — data abstractions. (The second most important issue is to identify the relations between these classes.)

In the Observer pattern, the key abstractions are “Publisher” and “Subscriber”. Both are useful concepts, but they turn out to yield an imperfect architecture; the basic reason is that these are not good enough abstractions for the publish-subscribe paradigm. At first sight they would appear to be appropriate, if only because they faithfully reflect the two words defining that paradigm; but what characterizes a good data abstraction is a set of consistent features. The only significant feature of a publisher is that it publishes events from a given event type, and the only significant feature of a subscriber is that it can subscribe to events from a given event type. That’s a bit light.

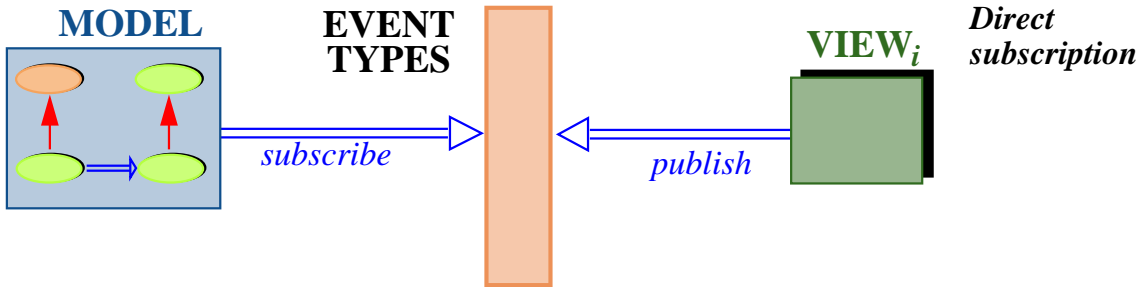
The more significant data abstraction, not recognized on its own right by the Observer design, is the notion of event type. It meets the criteria as it is a well-defined notion with a clear identity, and characteristic features: commands to publish and subscribe events, and the notion of argument (which could be given more weight through a setter command and a query).

Treating *EVENT_TYPE* as the key abstraction, yielding the basic class, enables us to avoid forcing publisher and subscriber classes to inherit from specific parents. A publisher is simply a software element that uses *publish* on a certain event type, and a subscriber a software element that uses *subscribe* for a certain event type.

MVC revisited

One of the consequences of the last design is to simplify the overall architecture suggested by the Model-View-Controller paradigm. The Controller part is “glue code” and it’s good to keep it to the strict minimum.

EVENT_TYPE provides the heart of the controller architecture. In a simple scheme it can actually be sufficient, if we let elements of the model subscribe directly to events:

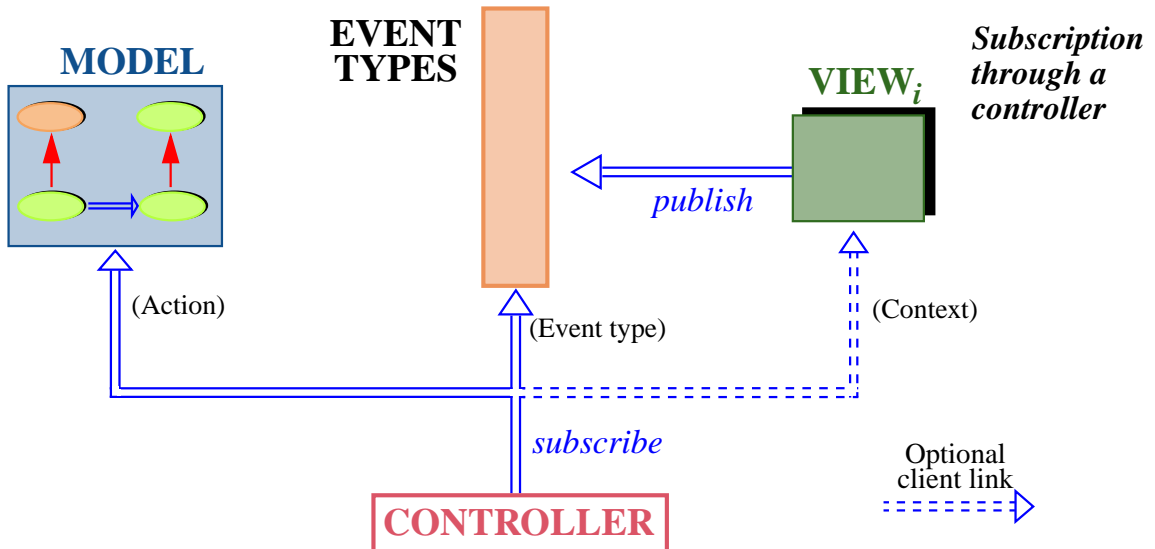


(The double arrows represent, as usual, the client relation, used here to implement the more abstract relations of the general MVC picture.) In this scheme there is no explicit controller component.

While the model does not directly know about the view (if it does not use contexts), it does connect to specific event types. This setup has both limitations and advantages:

- On the negative side, it can make it harder to change views: while we are not limited to a single view, any new view should trigger the same events. This assumes that the various views are not entirely dissimilar, for example a GUI and a Web interface.
- It has, on the other hand, the merit of great simplicity. Model elements can directly specify which actions to execute for specific events coming from the interface. There is essentially no glue code.

This scheme is good for relatively simple programs where the interface, or at least the interface style, is known and stable. For more sophisticated cases, we may reintroduce an explicit controller, taking the task of event type subscription away from the model:



The controller is now a separate part of the software. Its job is to subscribe elements of the model to event types; it will have connections to both:

- The model since the arguments to *subscribe* are actions to be subscribed, and these must be agents built from the mechanisms of the model.
- The view, if contexts are used. The figure shows this as an optional client link. ← As per style [3], page 525.

This solution achieves complete uncoupling between model and view; in a typical application the controller will still be still a small component, achieving the goal of using as little glue code as possible.

Invest and profit

Common to the two architectures we have seen, Observer and Event Library, is the need to subscribe to event types prior to processing them.

It is possible for subscribers to subscribe and unsubscribe at any time; in fact, with the Event Library solution, the program can create new event types at any stage of the computation. While this flexibility can be useful, the more typical usage scenario clearly divides execution into two steps:

- During initialization, subscribers register their actions, typically coming from the model.
- Then starts execution proper. At that stage the control structure becomes event-driven: execution proceeds as publishers trigger events, which (possibly depending on the contexts) cause execution of the subscribed model actions.

(So from the order of events it's really the "Subscribe-Publish" paradigm.)

Think of the life story of a successful investor: set up everything, then sit back and prosper from the proceeds.

You may remember another application of the same general approach, the "compilation" strategy that worked so well for topological sort: first translate the data into an appropriate form, then exploit it. ← "*Interpretation vs compilation*", page 446.

Assessing software architectures

The key to the quality of a software system is in its architecture, which covers such aspects as:

- The choice of classes, based on appropriate data abstractions.
- Deciding which classes will be related, with the overall goal of minimizing the number of such links, to preserve the ability to modify and reuse various parts of the software independently) but also choosing between client and inheritance.
- For each link, deciding between client and inheritance.
- Attaching features to the appropriate classes.

- Equipping classes and features with the proper contracts.
- For preconditions, deciding between a “demanding” style (strong preconditions, making the client responsible for providing appropriate values), a “tolerant” style (the reverse), or an intermediate solution.
- Removing unneeded elements.
- Avoiding code duplication and removing it if already present; techniques involve genericity, inheritance (making two or more classes inherit from a common ancestor that captures the common elements).
- Taking advantage of known design patterns.
- Devising good APIs: simple, easy to learn and remember, equipped with the proper contracts.
- Ensuring consistency: throughout the system, similar goals should be ensured by similar means. This governs all the aspects listed so far; for example, if you use inheritance for a certain class relationship, you shouldn’t use client elsewhere if the conditions are the same; consistency is also particularly important for an API, to ensure that once programmers have learned to use a certain group of classes they can expect to find similar conventions in others.

Such tasks can be carried out to improve existing designs, an activity known as *refactoring*. It’s indeed a good idea always to look at existing software critically, but prevention beats cure. The best time to do design is the first time.

Whether it’s done as initial design or as refactoring, work on software architecture is challenging and rewarding; the discussion in this chapter — and a few others in this book, such as the development of topological sort — give an idea of what it involves. The criteria for success are always the same:

Touch of Methodology: **Assessing software architectures**

When examining possible design solutions for a given problem, discuss alternatives critically. The key criteria, are: reliability, extendibility, reusability, and simplicity.

20.7 FURTHER READING

Bertrand Meyer: *The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design*, in *From Object-Oriented to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271. Available online at se.ethz.ch/~meyer/publications/lncs/events.pdf.

A significant part of the present chapter's material derives from this article, which analyzes the publish-subscribe pattern in depth, discussing three solutions: Observer pattern, .NET delegate mechanism, and the event library as presented above.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns*, Addison-Wesley, 1994.

A widely used reference on design patterns. Contains the standard description of Observer, along with many others, all expressed in C++.

Trygve Reenskaug, MVC papers at heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

Trygve Reenskaug, a Norwegian computer scientist, introduced the Model-View-Controller pattern while working at Xerox PARC (the famed Palo Alto Research Center) in 1979. The page listed contains a collection of his papers on the topic. I find his original 1979 MVC memo (barely more than a page) still one of the best presentations of MVC.



Reenskaug

20.8 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Event-driven design, also called “publish-subscribe” leads to systems whose execution is driven by responses to events rather than by traditional control structures. The events are triggered by the software, often in reaction to external events. GUI programming is one of the important areas of application.
- The key abstraction in event-driven design is the notion of event type.
- *Publishers* are software elements that may trigger events of a certain event type. *Subscribers* are elements that request to be notified of events of a certain type by *registering* actions to be executed in response.
- In a system with one or more interfaces or “views”, an important design guideline is to keep the views separate from the core or the application, known as the “model”.
- The Model-View-Controller architecture interposes a “controller” between the model and the view to handle interactions with users.
- The Observer pattern addresses event-driven design by providing high-level classes *PUBLISHER* and *SUBSCRIBER*, from which publisher and subscriber classes must respectively inherit. Every subscriber class provides an *update* procedure to describe the action to be executed in response to event. Internally, each publisher object keeps a list of its subscribers. To trigger an event, it calls *update* on its subscribers; thanks to dynamic binding, each calls the proper version.
- Agents, constrained genericity and tuples allows a general solution to event-driven design through a single reusable class based on the problem's central abstraction: *EVENT_TYPE*.

- Software architecture is the key to software quality. Devising effective architectures, and improving existing ones (refactoring) should be a constant effort, focused on simplicity and striving at reliability, extendibility and reusability.

New vocabulary

Application domain	Argument (of an event)	Business model
Catching (an event)	Context (of an event)	Control (Windows)
Controller	Event	Event-driven
Event type	External event	Glue code
Handling (an event)	Model	MVC
Publish (an event)	Publish-Subscribe	Register
Refactoring	Signature (of event type)	Subscribe
Trigger (an event)	View	Widget

20-E EXERCISES

20-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

20-E.2 Efficient Observer

Choosing the appropriate representation of the subscribers list, adapt the implementation of the Observer pattern so that the following operations are all **O(1)**: add a subscriber (doing nothing if it was already subscribed); remove a subscriber (doing nothing if it was not subscribed); find out if a potential subscriber is subscribed. The *publish* procedure, ignoring the time taken by subscribers' actual handling of the event, should be **O(count)** where *count* is the number of subscribers actually subscribed to the publisher. Overall space requirement for the *subscribers* data structure should be reasonable, e.g. **O(count)**. (*Hint*: look at the various data structures of chapter [10](#) and at the corresponding classes in EiffelBase.) Note that this optimization also applies to the Event Library implementation.

← ["THE OBSERVER PATTERN", 20.4, page 515.](#)

20-E.3 Type-safe Observer

Show that in implementing the Observer pattern a type scheme is possible that removes the drawbacks of both “[Argument Scheme 1](#)” and “[Argument Scheme 2](#)” by taking advantage, as in the last design of this chapter (Event Library), of tuple types and constrained genericity. Your solution should describe how the *PUBLISHER* and *SUBSCRIBER* classes will change, and also present a typical publisher and subscriber classes inheriting from these. ← [“The subscriber side”, page 518.](#)

21

Program correctness

21.1 DESIGN BY CONTRACT

21.2 HOARE TRIPLES

21.3 ASSIGNMENT RULES

21.4 RULES ON CONTROL STRUCTURES

21.5 AN EXAMPLE PROOF

PART V:

Towards software engineering

W

22

Overview of software engineering

There is more to software development than programming. This statement is not a paradox, but a recognition of all the factors that affect the success of a software project, and all the resulting tasks that we must accordingly worry about, in addition to writing the program. To take just a few examples:

- A program with a brilliant design may end up a failure if its user interface displeases the target audience.
- The best program is useless if it doesn't solve the right problem. Hence the need for a *requirements* task to capture user needs and decide on the system's precise functionality.
- Aside from technical aspects, projects must tackle *management* issues: setting and enforcing deadlines, organizing meetings and other communication between project members, defining the budget and controlling expenses.

These activities and many others discussed in this chapter do not involve programming techniques, but if not taken care of properly they can destroy a project regardless of its technical qualities.

This is typical of what defines moving beyond *programming* to *software engineering*. In the previous chapters we have almost exclusively been concerned with programming, but the picture would be incomplete without a foray into the non-programming aspects of software engineering.

This is a wide-ranging and well-developed discipline. To cover it extensively would mean another textbook, such as those cited in the “Further reading” [section](#). The present chapter has more limited goals: to present a general survey, enough I hope to awaken your interest and make you want to learn further from the rest of the literature. → [Page 556](#).

22.1 BASIC DEFINITIONS

The following broad-ranging definition will serve us well:

Definition: Software engineering

Software engineering is the set of techniques — including theories, methods, processes, tools and languages — for developing and operating production software meeting defined standards of quality.

Two important properties of software engineering captured by this definition are the restriction to production software, and the focus on quality.

Production software is operational software, intended to function in real environments to solve real problems. Software developed purely as an experiment, or “throw-away” programs used once and not further maintained, generally do not qualify, except if they are a means towards some broader goal which belongs to software engineering proper. For example, an experiment to evaluate various possible algorithms may not qualify by itself, but this changes if it is performed as part of the development of a production system. Similarly, textbook examples are usually not software engineering — except if they are designed specifically to illustrate techniques applicable to production systems, or are extracted from such a system.

What characterizes production software is the combination of constraints that it must satisfy. They may include:

- Quality constraints as discussed next; for example the guarantee that the system will not crash, will deliver correct results, will perform fast.
- Size constraints: production systems may consist of thousands or tens of classes and other modules, and hundreds of thousands or millions of lines of code.
- Duration constraints: systems used in industry must often be maintained (that is to say, kept operational, and regularly updated) over many years or even decades.
- Team constraints: such systems may involve large teams of developers, and large numbers of users; this raises specific management and communication problems.
- Impact constraints: these systems affect physical and human processes; in particular, if they do not function well, people may be affected — by a train not arriving in time, a phone not working, a salary not paid, an order not delivered, or worse. This reinforces the emphasis on quality.

Quality is indeed at the center of software engineering concerns. The definition mentions “defined standards”: quality is not just something that someone claims exists, or doesn’t, in a software process or product; it should be evaluated as objectively as possible.

The definition also talks of “developing and *operating*” software. Software construction cannot be hit-and-run: along with development you have to set up the actual operation. Even the development part should not be understood as only the initial production of a releasable system: what comes afterwards is just as important. We have already encountered the technical term for this activity:

Definition: Software maintenance

Maintenance of software systems covers all further development activities occurring after the first release of an operational version, such as: adaptation to new platforms and environments; correction of reported deficiencies; extensions (addition of new functionality); removal of unneeded functionality; quality improvement.

The term “maintenance” comes from other parts of engineering: think of maintenance for a car, a coffee machine, a house. It is often pointed out that the analogy is misleading, since a program doesn’t deteriorate from repeated use; run your program ten, a thousand or a million times, and unlike a car whose tires will inexorably wear out it’s exactly the same program as the first time. As a software term, however, “maintenance” is here to stay and there’s no problem in using it as long as it’s based on a precise definition as above.

A jargon term will be useful for the discussion:

Stakeholder

A **stakeholder** of a software project is any person who can affect or be affected by the project and the quality of the resulting software.

This encompasses many people: developers, but also testers and other quality assurance personnel, project managers, future users of the system (or others on whom it may have an effect, including — the less pleasant part but definitely a possibility — those who will *not* be users because the system makes their current jobs obsolete), marketing and sales people who will have to find customers in the case of a product to be released to the world, trainers (who will educate users), corporate legal departments. It is an important task of project management to identify all the stakeholders early and to give due consideration to the needs and constraints of each.

22.2 COMPONENTS OF QUALITY

Quality, the central pursuit of software engineering, is a notion with many different components, often called *factors* of software quality. Let's take a look at some of the most important ones.

Process and product

Issues of software engineering involve two complementary aspects:

- *Products*: outcomes of the development. The most obvious product is the source code, but significant software projects adds many others such as requirements and design documents, test data, project plans, documentation, installation procedures.
- *Process*: mechanisms used to obtain these products.

The number and severity of errors in a delivered program is an example of a product issue. Whether the program is delivered on schedule is an example of a process issue.

In each case the other aspect plays a role too: the process determines in part the introduction and removal of errors; and treating timely delivery as the principal goal may affect the product, for example through dropped functionality.

It is convenient to discuss the factors of software quality under three rubrics based on this distinction:

- *Process quality*, characterizing the effectiveness of the software development process.
- *Immediate product quality*, characterizing the adequacy of the product as delivered in a particular version.
- *Long-term product quality*, characterizing the future prospects of the software. In the world of software engineering, where projects may have a long life, this is just as important as the immediate picture.

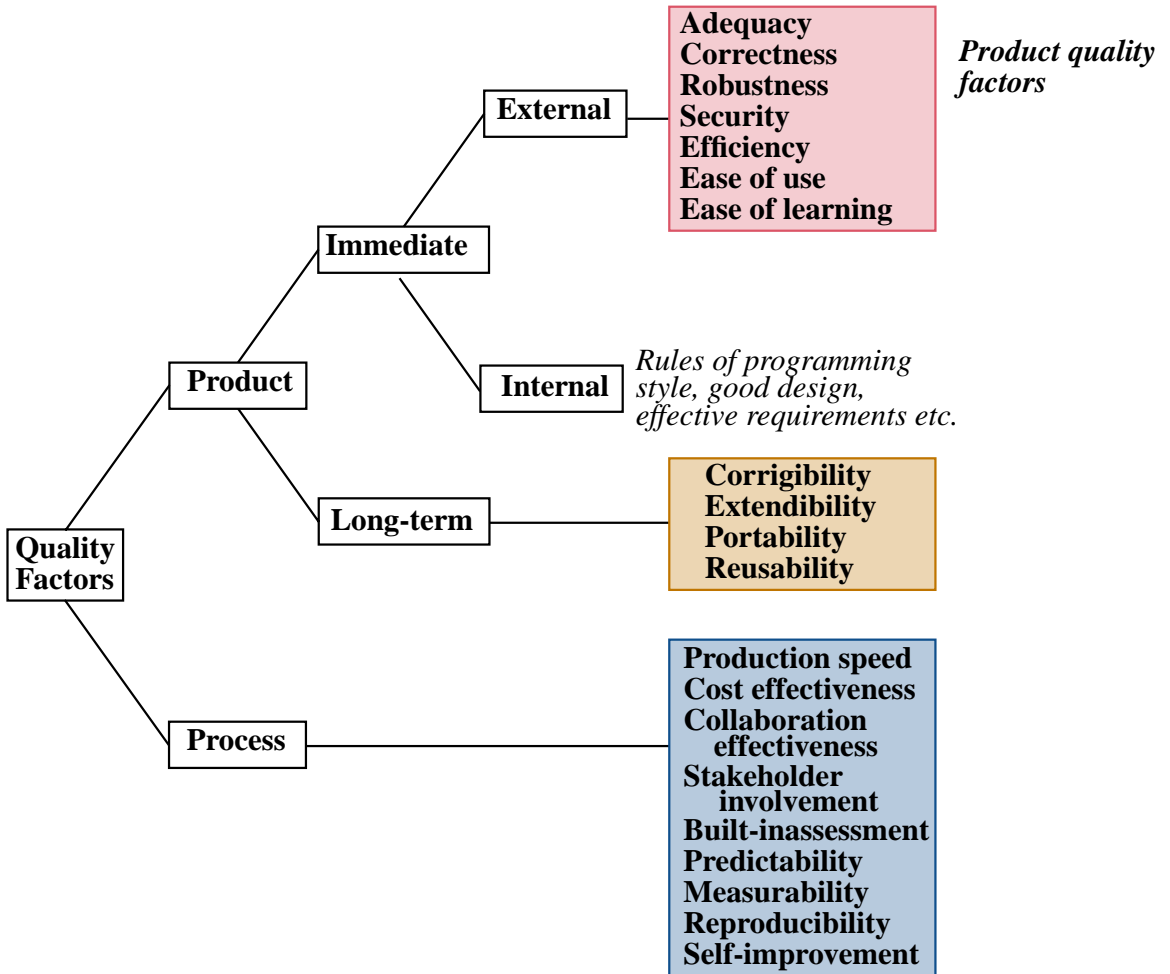
We will now take a look at the major goals in each area, starting with the most visible property of a software project — immediate product quality. The discussion also includes some comments about why some other factors are less relevant. Two general notes about this review:

- No explicit definitions are given for self-explanatory quality factors (“ease of use”, “ease of learning”). The corresponding terms will not appear in the “New vocabulary” [list](#) of this chapter.
- You will notice a certain relativism in the definitions: adequacy is satisfaction of *defined* needs, efficiency is *adequate* use of resources. This is not vagueness but in fact the reverse: definitions of software quality goals are only useful inasmuch as they allow the product or process to be **assessed** against these goals. The definitions consequently assume that such goals have been clearly defined. This issue is not just academic:

→ Page [558](#).

imagine you are heading a software development project and that you track the number of remaining deficiencies (“bugs”). Should you OK the release when the number reaches 1000, 500, 200, 0? (In a realistic setup you would have to distinguish between categories of bugs: critical show-stopping problems, minor issues such as user interface imperfections, “nice to have” missing functionalities that could be deferred to the next release and so on.) This question is essentially impossible to answer unless precise criteria have been stated in advance. We are back to the original definition of software engineering and its insistence on “meeting *defined* standards of quality”. ← Page 540.

The following figure shows the overall classification for the quality factors to be reviewed now.



Immediate product quality

Product quality involves the following factors.

- **Adequacy:** satisfaction of defined user needs. In other words: does the software serve the right purposes for its user community? Other factors commonly cited in this area are *completeness* and *usefulness*, but both are less precise and are subsumed by “adequacy”: no system ever has “complete” functionality, since someone will always think of another facility that would be nice to have; and “usefulness” is a subjective criterion unless you state precisely *to whom* and *for what needs* the system is, or not, useful enough.
- **Correctness:** to what extent the software functions as prescribed by the specification, in cases covered by the specification. This is clearly a fundamental requirement. It is just as clearly hard to achieve, not only because writing programs that meet a specification is hard, but also because writing the specification itself is tough too — you must think of all cases and end up with a document that is precise yet readable.
- **Robustness:** how well the system reacts to erroneous cases of use, outside of the specification. That a user pressed the wrong button, a sensor malfunctions or another program sent bad input is not a good excuse for the system to crash or produce wild results. Robustness assesses error handling and recovery mechanisms.
- **Security:** how well the system protects itself, its data, its users and any affected devices or people against hostile attempts at misuse. Unfortunately it’s not just errors we need to worry about, as addressed by robustness; computer systems offer ready targets for people with all kinds of nasty intent, and you cannot write software, especially if it will be available over a network, without thinking of potential attacks.
- **Efficiency** (often called **performance**): adequate use of time, memory space and other resources such as bandwidth if the system engages in network communication. “Adequate”, not optimal: if your compiled program takes up one megabyte of memory, reducing this to 0.6 MB may be possible, but is not necessarily useful. If you expect your users to have plenty of memory, it is probably more productive to spend your time on other quality factors; but if you are running in a tightly constrained environment, for example with software for small-memory handheld devices, such space optimization can become critical. What matters once again is to define objectives.

- **Ease of use.** The really difficult challenge here is to make the system easy to use for *various categories* of users. Much of the effort in “usability”, as this is also called, goes into facilitating the task of complete novices. But it’s just as important to help the experts — who, for example, don’t want to go through the same repetitive sequences of clicking “OK” on various informational windows over and over again, when they know exactly what to do — and to support the process of progressing from novice to expert. Each of us is a novice for some tools and an expert in others; and each of us, for each of the system in which we are an expert, was a novice once. Ease of use is also about defining that path and helping anyone who wants to travel it.
- **Ease of learning**, closely connected to the previous factor.

Long-term product quality

Some product qualities are of no immediate value to users of a system, but of much interest to those who commission or purchase it. If I am driving, I don’t care that the software controlling brakes or the air bag is easy to modify; I care that it works (an “immediate” factor). But if I am an executive in charge of managing software development or acquisition at Nissan or BMW I have to keep in mind the long-term picture: will the software be easy to upgrade if an improvement is requested? Can the version developed for sedans be transposed at reasonable cost to convertibles?

Descriptions of software products and software issues often talk about “the user”; the term that has acquired almost mythical connotations. It’s good to think of users, but stakeholders in user organizations also include others with a long-term perspective. The more general term “customer” is appropriate (whether or not the product is commercial) to cover both people using the products now and those interested in its past and future.

Long-term qualities, in the approximate order of when concerns will arise, include:

- **Corrigibility:** how easy it is to update the software to repair deficiencies (of correctness, robustness, security, ease of use...). One of the recipes for achieving corrigibility is *structure*: devising a modular architecture that is easy to understand and reflects the structure of the problem and its solution. *Also “correctibility”.*
- **Extendibility:** how easy it is to add functionality. Here too, structure is key; the object-oriented techniques we have learned — data abstraction, information hiding, classes, genericity, contracts, inheritance, dynamic binding, agents and so on — facilitate extension. Extendibility is a principal requirement of practical software development, as almost every system undergoes changes of its expected functionality. The reason for *Also “extensibility”.*

change may be that the initial requirements definition missed some functions; sometimes it is simply the consequence of initial success, as a useful system suggests ideas of what more it could do. A good software process must enforce a discipline on such changes, by defining strict procedures for examining new requests once initial requirements have been approved; but it cannot pretend that the need will not arise.

- **Portability:** how easy it is to transfer the software to other platforms. A “platform” here is a combination of computer architecture and operating systems, plus other resources that the system may need, such as a database management system. The IT industry has experienced considerable standardization in recent decades, making the construction of portable software more realistic than when dozens of incompatible computer brands populated the market. For general-purpose computing, the hardware scene is down to a few architectures (Pentium and compatible, Sparc), and the operating system world to Windows and to Unix variants such as Linux, Solaris and MacOS. As to programming language, C, Java and Eiffel are available on numerous platforms.
- **Reusability:** how much of the product can be applied to future developments. Many applications need some of the same functionality, either of a general nature (data structures and fundamental algorithms, GUI mechanisms) or targeted to a particular application domain. *Reusable software* is software that is sufficiently independent from the specifics of a particular project to be of use for subsequent ones. Helped by object technology, reusability in software has made great strides, leading to **software components** that serve the needs of many different developments. (Think of the Traffic library and all the libraries on which it itself relies.) Even if you are not producing software components you can strive to make your software reusable to facilitate future projects.

In the literature you will see references to a quality factor called **maintainability**, having to do with the ease of continuing to work on a system after its initial release. This important concern is not an independent factor but a combination of the long-term product factors just reviewed, as maintenance may involve fixing errors, adding functionality and adapting to new platforms.

← “*Definition: Software maintenance*”, page 541

Other factors that do not figure in the above list are what we may call **internal** product quality factors. You informally know many of them because they correspond to the programming advice given throughout this book, telling you to ensure your software is well-structured and readable, applies information hiding, uses contracts and so on. Another example of internal factors is the list of properties defined below for good *requirements* documents, some of which also apply to programs. Internal qualities are fundamental attributes of a software system; so fundamental indeed that only through them can external factors be achieved. Correctness and corrigibility, for example, both come down to matters of systematic programming, good structure and having the right contracts.

From the global perspective of software engineering, the relevant product factors are the **external** ones just discussed as they are relevant to customers. Internal factors belong to the technology of programming.

Process quality

Process factors address the quality of the mechanisms used to produce the software. They include the following:

- **Production speed:** the ability to deliver a product in a short time. Every project has to worry about this; customers are waiting, competitors progressing, shareholders wondering.
- **Cost effectiveness.** This is also a concern for almost all projects. In software (unlike some other fields of engineering) the production cost is usually negligible. *Development cost* dominates everything else (except sometimes the cost of marketing, which can be significant especially for mass-market products); within it, *personnel costs* dominate other aspects such as equipment and office space. For that reason the standard measure of cost is the *person-month*: the average cost of employing one person — employee, contractor — for one month, all-inclusive.
- **Collaboration effectiveness:** the effectiveness of procedures for combining the contributions of all project members and allowing them to communicate. Significant software projects may involve large numbers of people, requiring special attention to coordination mechanisms. Communication in particular is a delicate issue, which beyond a certain team size can overshadow all other aspects of the development. An extreme form of this phenomenon is known as “Brooks’s Law” (from the name of the designer of the IBM OS-360 operating system), which states that adding people to a late project delays it further. This is only true of badly managed projects but highlights the need to devote proper attention to communication issues.

Often also
“man-month”.

→ See book citation in
“[FURTHER READ-
ING](#)”, 22.8, page 556.

- **Stakeholder involvement:** the degree to which the project takes into account all relevant needs and viewpoints.
- **Built-in assessment:** the inclusion in the process of mechanisms and procedures to evaluate product and process quality factors at well-defined steps. Quality is not just decreed and attempted, it must be checked and enforced. A good process integrates this task as one of its components.
- **Predictability:** the inclusion in the process of reliable methods to estimate other quality factors — in particular production speed and cost — ahead of time,. Predictability is one of the most important characteristics of a good process; sometimes a guaranteed date is just as important as an early date. The software industry has not had a good record in this area, as many projects are late and over budget; the situation is improving, thanks to better application of software engineering principles and techniques.
- **Measurability:** the availability of sound quantitative criteria to determine achievement of other quality factors, both process and product; for example, techniques for measuring error rates. Effective management needs precise measures of progress. This criterion is closely related to the preceding two, since the ability to predict and to assess whether the prediction was met requires the ability to measure.
- **Reproducibility:** the independence of development, management and prediction techniques from unessential attributes of individual projects. In most industrial contexts a software development doesn't happen in isolation but as one in a succession of projects. It is important to carry over information and experience from one project to others, so that a success in one particular project (reproducibility of *failures* is not attractive) can be replicated on future ones. This means in particular being able to abstract process and product attributes from the circumstantial properties of particular projects, such as the personalities of the developers and the specifics of the customer. Such reproducibility is one of the characteristics of an industrial production process. Because software is an intellectual activity, not assembly-line production, no one will ever achieve total reproducibility, nor would it necessarily be desirable; but a good software process reduces unnecessary sources of non-reproducibility — bad surprises.
- **Self-improvement:** the inclusion in the process specification of mechanisms to qualify and improve the process itself. Organizations, like people, can learn from experience. The self-improvement criterion assesses to what extent the process, as defined by the organization, encourages this phenomenon by including built-in evaluation mechanisms, which can be fed back into the process itself for adapting it a result of the lessons learned.

Process models such as CMMI studied next take these issues to heart, in particular the last five, to foster a software culture in which built-in assessment, predictability, measurability, reproducibility and self-improvement are built in as core practices.

Tradeoffs

While any software development should strive for the highest quality in all respects, the preceding review shows that tradeoffs are inevitable:

- Tradeoffs between process and product factors: a quest for perfection in the product might take too long to achieve, affecting “production speed”.
- Tradeoffs between product factors: ease of use doesn’t always agree with security, since you will only want to make the product easy for *legitimate* users. Passwords are bad for ease of use but good for security; optimizing for efficiency can conflict with corrigibility (as it may lead to contorted code), and with factors such as extendibility, portability and reusability, all of which call for general solutions rather than techniques narrowly targeted to a particular platform and context.

One of the characteristics of a well-managed project is that it examines these tradeoffs explicitly, and resolves them on the basis of rational analysis. Otherwise they end up being resolved anyway, but not necessarily in the most desirable way; a common example is a misplaced concern for efficiency — extensive optimization where it’s not essential— obtained at the expense of other quality factors.

22.3 MAJOR SOFTWARE DEVELOPMENT ACTIVITIES

Software engineering involves a number of tasks. You have learned much about one of them, implementation, and gained a good first idea of others such as design, documentation, specification. Let’s go through the list of major tasks; the order is, roughly, from tasks closest to customers’ concerns to those dealing with technical software needs.

Feasibility analysis is the task of studying a customer-related problem and deciding if it’s possible and desirable to build a software system, or a system involving software, to address it. The second aspect, although not immediately suggested by the name, is just as important as the first; not every system that can be built should.

Requirements analysis defines the functionality of the system. The elements making up a requirements document are of two kinds:

- *Functional* requirements, describing the results or actions of the system: “If the phone user leaves a coverage area to enter another, the connection shall switch automatically to an access point in the new area”.

- *Non-functional* requirements, specifying constraints on the system's operation. They include *performance* requirements such as timing (“For an access point less than two kilometers away, switchover shall take no more than one second”), memory and bandwidth usage, security (“all communication with the access point shall be encrypted”); they also cover impact on the system's environment and consequences for stakeholders such as employees (effect on work practices, training requirements).

Specification is the precise description of individual elements of the system. Requirements are customer-oriented; specification translates requirements into a form that is directly usable for the development of the software. The main difference is rigor and precision: the specification must give an unambiguous answer to every relevant question about the operation of the system.

Requirements and specification are sometimes treated as a single activity; the world **analysis** is then used to cover them both. In the lifecycle models that follow we will treat them as separate. Regardless of the exact division, the activities seen so far only address the *problem* to be solved; with the next tasks we enter the world of software *solutions*.

Design, also called **architecture**, builds the overall structure of a software system. It is responsible in particular for defining the principal units, or *modules*, of that system, and the relations between those units.

Implementation is the task of actually developing the program text to produce a usable system. This is also known as *coding*, with just a hint of a derogatory tone — as if writing the program were a menial chore to be performed once the great thinkers have done the analysis and design. (In this book *programming* is used in the broad sense of program construction: not just implementation but also design and analysis.)

Documentation is the task of describing various aspects of the system to help its users and other stakeholders, in particular developers. Aside from documents for users it may include project plans (for managers) and documents describing the results of some of the other tasks: requirements documents, specifications, design plans. The word “document” encompasses more than traditional reports designed for paper; today's documentation takes many other formats such as Web pages, online help files, or explanations included in program texts and processed by specialized tools (such as the header comments in Eiffel classes, or, in Java programs, special comments marked as “Javadoc”).

Validation and Verification, or “V&V”, is the task of assessing whether the system is satisfactory. The two aspects are complementary:

- Verification is *internal* assessment of the consistency of the product, considered just by itself. A typical example, at the level of implementation, is type checking, preventing you for example from declaring a variable as *REAL* and using it as if it were an *INTEGER*.

- Validation is the *relative* assessment of a product vis-à-vis another that defines some of the properties that it should satisfy: code against design, design against specification, specification against requirements, documentation against standards, observed practices against company rules, delivery dates against project milestones, observed defect rates against defined goals, test suites against coverage metrics.

A popular version of this distinction is that verification is about ascertaining that the product is “doing things right” and validation that it is “doing the right thing”. It only applies to code, however, since a specification, a project plan or a test plan do not “do” anything.

“Maintenance”, as already noted, is not a separate activity but a combination of some of the tasks listed above.

22.4 LIFECYCLE MODELS

22.5 VERIFICATION AND VALIDATION

22.6 CAPABILITY MATURITY MODELS

Assume you are in an organization that needs to contract out some development to a software company. There’s no product yet to judge, so all you can evaluate is the process. The company tells you they have everything under control, but how do you know?

In the early nineties this need for objective assessment of companies’ software processes led the US Department of Defense (DOD), the world’s largest consumer of software services, to ask the Software Engineering Institute, a DOD-funded center at Carnegie-Mellon University in Pittsburgh, to develop a model for the level of industrial “maturity” of software organizations. The resulting “Capability Maturity Model”, further developed into a more comprehensive set of models known as CMMI (“I” for “Integration”), has exerted a profound influence on several segments of the software industry, in particular:

- US defense contractors, its initial target.
- Indian software companies, probably not part of the initial plan; India’s nascent outsourcing industry saw in the CMM, as it was then called, a critical tool for obtaining outside certification that would reassure the Western customers they were trying to attract. Soon after the model was released, Indian companies started to account for a significant share of CMM certifications.

CMMI is also used outside of these communities. As a sign that it has extended its reach beyond its initial target group, the proportion of defense contractors and military organizations performing CMMI assessment went down to 40% in 2004 and continues to decrease.

Some companies interested in process improvement and qualification prefer other models, in particular the 9000 series of standards from the International Standards Organization (ISO), the software-oriented branch of a set of international standards for industrial quality in general, and SPICE (Software Process Improvement and Quality dEtermination) which combines some elements of the other two. In this overview we'll just look at CMMI.

CMMI scope

CMMI and consorts examine only the process. They are technology-neutral, language-neutral, tool-neutral. All they assess is whether the organization has a set of clear procedures in place, applies them, controls that it applies them, measures their effect, and strives to improve them. In terms of the preceding discussion of software quality, the emphasis is on the *process* factors, especially the last five on our list. Think of the pilot and copilot going through their check-list prior to a flight: what matters is that they consider every single item on the list, tick it off if it's OK, and follow the predetermined action (such as calling aircraft maintenance) if not. Because of this emphasis on formal procedures at the expense of technology, some people dismiss process models as merely a way for managers to “cover their bottoms”, in case the project doesn't fare well, by showing that they did everything by the book. Indeed there have been cases of major project failures in organizations with high CMMI or ISO qualifications. But such dismissal is a classical case of confusing necessary with sufficient: software projects, especially large ones, need both high process quality and high product quality. You can still mess up with a perfect process, but process qualification is one among a set of tools available to companies to help them *not* mess up.

Key to CMMI is the notion of assessment. Organizations wishing to establish their “maturity level” as discussed next may get themselves evaluated — in the military's passionate acronym culture this yields an example of nesting, SCAMPI for “*Standard CMMI Appraisal Method for Process Improvement*” — by assessors officially accredited by the Software Engineering Institute: 179 “SEI partners”, organizations rather than individuals, as of 2005. Assessed organizations may publish the results of the assessment — typically, to boost their attractiveness if they are software companies — or keep them for themselves. Between April 2002 and September 2004, the SEI was notified of 424 appraisals affecting 206 companies, half of them outside the US.

CMMI disciplines

As the I in the acronym attests (“Integration”), CMMI outgrew the original CMM to cover a range of models that extend beyond software; the four “disciplines” covered include software engineering but also:

- “Systems engineering”. This concept covers non-software aspects of a system; indeed, software is often part of a bigger system — think of the software on your cell phone — which has its own process, involving hardware, software and other aspects.
- “Integrated product and process development”.
- “Supplier sourcing”: selecting, controlling and coordinating all the suppliers that contribute to a project. Large projects often involve the participation of many suppliers; in some cases, for example a government customer with no software development department of its own, a project is entirely outsourced. Supplier sourcing covers oversight of the outsourced work.

An organization interested in implementing CMMI and getting assessed may select from these disciplines, depending on its activity and needs.

Goals, practices and process areas

The essence of CMMI is to define **goals** and recommend **practices**:

- A goal is a desirable property of a process. For example, every project should have good requirements, describing user needs; this observation yields goals such as “*Develop customer requirements*” and “*Analyze and validate requirements*” (that is to say, it’s not enough just to produce requirements for a project, but one should also have formal procedures to check that they are feasible and satisfy the stakeholders).
- A practice is a technique that has been shown to help achieve a goal. Examples are “*Establish a definition of required functionality*” and “*Analyze requirements to establish balance between stakeholder needs and [project] constraints*”.

As the examples indicate, every practice must be related to a certain goal; using software terminology, the goal is a specification, the practice an implementation (carried out by humans) of that specification.

Such goals and the corresponding practices are grouped into collections called **process areas**. The preceding examples are part of the process area “*Requirements development*”.

The term “area” is not intuitive, so to understand the rest of the discussion you must remember that a “process area” is exactly what this definition says: a collection of goals and of practices supporting those goals.

Two models

CMMI exists in two variants: *staged* and *continuous*. The difference is scope:

- The staged variant addresses the maturity level of an organization as a whole. This has the merit of yielding a single, global figure (“Our division just achieved CMMI qualification at level 4!”) but ignores the differences between various activities and specialties; for example an organization might be very good at software construction but not have mastered requirements yet. Staged description is in the tradition of the original CMM, and is still the dominant practice.
- Continuous description allows assessment of individual process areas and hence provides more flexibility.

Common to both variants is the notion of assessment level. CMMI enables you to qualify your organization — all of it if staged, some of its process areas if continuous — at one of **five levels**, labeled 1 to 5 in order of increasing closeness to the Nirvāna of total control. (The continuous representation adds a level 0, “incomplete”, for process areas not applied.)

In the staged variant, each level is characterized by a set of process areas: you reach that level if you apply the corresponding practices and satisfy the corresponding goals. For example, reaching level 2 assumes that you satisfy *Requirements management* and other process areas listed below. In addition, each level has one **generic goal** and a corresponding set of **generic practices** not belonging to any process area; for example level 2 has the generic goal “*Institutionalize a managed process*”, meaning a company-wide definition and enforcement of a development process, and associated generic practices such as “*Plan the process*” and “*Provide resources*”.

As a consequence of this concept the goals and practices are divided into two categories:

- **Generic**: characterizing a CMMI level, but not belonging to a particular process area.
- Those belonging to a process area, called **specific**.

Assessment levels

Here is the general characterization of the levels, in the staged variant. The more precise definition comes from the table below, which identifies the generic and specific goals of each. There are, as noted, five levels:

- 1 • **Initial:** this characterizes an organization with little process definition or enforcement. Some projects succeed, others not, but no one quite knows the reason. It’s like going for mushrooms in the woods on a rainy day in October: this oak has lots, that one has none, but why? To me they look just the same. In software development this is sometimes known as the “heroic” stage: success depends too much on the people involved and on the poorly controlled circumstances of each project.
- 2 • **Managed:** at this level there is a real process; the organization has defined policies which include a description of the process and plans for carrying it out; it has allocated resources and defined responsibilities to meet these plans; application of the process is monitored, reviewed, and reported to higher management; stakeholders are defined and involved; there’s a mechanism for configuration management (a topic discussed below). In other words, the process has been defined and is carefully carried out.
- 3 • **Defined:** this is a managed process (from now on each level assumes the preceding ones) with more systematic procedures. The main difference with the previous level is the mix of *generality* and *tailorisation*: there is a global but customizable process model for the organization as a whole, and the process for any project is customized from it.
- 4 • **Quantitatively managed:** in addition to the previous requirements, the process makes extensive use not only of quantitative data (such as measures of costs, development times, reliability, service quality indicators) but of statistical quality control techniques to analyze the data in depth and use the results as part of the process.
- 5 • **Optimized:** adds a feedback loop that uses data collected about the projects to question the process and improve it continually, both incrementally and through more innovative changes.

Not to be confused with Taylorisation (which is how critics would characterize the whole thing).

The following table describes, more precisely, what must be achieved at each level (starting at 2 since by definition there’s nothing at level 1).

Level	Name	Generic practices	Process areas
2	Managed		Requirements management Project planning Project monitoring & control Supplier agreement management Measurement & analysis Process & product quality assurance Configuration management

3	Defined		Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management for IPPD Risk management Integrated teaming Integrated supplier management Decision analysis & resolution Organizational environment for integration
4	Quantitatively managed		Organizational process performance Quantitative project management
5	Optimized		Organizational innovation & deployment Causal analysis and resolution

Goals and practices for each level

22.7 FURTHER READING

Software Engineering Institute: Capability Maturity Model Integration (CMMI) Overview , online document at www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview05.pdf.

A short overview of CMMI, in the form of presentation slides.

Software Engineering Institute: *Capability Maturity Model® Integration (CMMISM), Version 1.1, CMMISM for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1) Staged Representation CMU/SEI-2002-TR-012 ESC-TR-2002-012* (sorry, I don't make those titles). Available online at tinyurl.com/kf9uy (shorthand for www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf#search=%22cmmi%20staged%20representation%22).

This is the official, detailed description of CMMI, staged representation. (Continuous variant at tinyurl.com/gjla9; the two documents share a large amount of material.) You will need to gear yourself up for the discreet charm of Government-Committee English, probably not quite what your creative writing instructor has in mind when she advocates conciseness, concreteness and clarity. A sample: “*The plan for performing the organizational process focus process, which is often called ‘the process-improvement plan,’ differs from the process action plans described in specific practices in this process area. The plan called for in this generic practice addresses the comprehensive planning for all of the specific practices in this process area, from the establishment of organizational process needs all the way through to the incorporation of process-related experiences into the organizational process assets*” — Wow!. Once you’ve learned to ignore the bureaucratise you will in fact find, like little gems in the rubble, a concentrate of some of the best project organization practices having emerged from four decades of software project management experience.

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli: *Fundamentals of Software Engineering, 2nd Edition*, Prentice-Hall, 2002.

A good software engineering textbook, providing broad coverage of the field. Other useful textbooks are by: S.L. Pfleeger and J. Atlee (3rd edition, Prentice Hall, 2005); and Roger Pressman (6th edition, McGraw Hill, 2005).

Frederick P. Brooks: *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*, Addison-Wesley, 1995 (the original edition is from 1975, same publisher).

At IBM Fred Brooks directed the development of OS-360, one of the first example of a complex operating system available across a whole line of computers. This book, where he summarized his experience through short individual essays, has to be mentioned here since it is widely considered a classic in software engineering, although it’s more for its folksy advice than for any deep contribution.

22.8 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Software engineering encompasses programming but also all the other activities, technical or not, involved in producing quality software systems. It focuses on industrial software production with defined standards of quality.

- Issues of software engineering affect both the development *process* and the resulting *products*.

New vocabulary

Adequacy	Built-in assessment	Correctness
Correctibility	Cost control	Efficiency
Extendibility	Factor (of software quality)	Goal (CMMI)
Lifecycle	Maintenance	Measurability
Portability	Practice (CMMI)	Predictability
Process (vs product)	Process area (CMMI)	Product (vs process)
Production software	Reproducibility	Reusability
Robustness	Security	Self-improvement
Software engineering	Stakeholder	

The names of some of the quality factors (ease of use, production speed...) retain their meanings from non-technical usage and do not figure in this list.

← "[*COMPONENTS OF QUALITY*](#)", 22.2, page 542

Acronym collection

CMM	CMMI	DOD
ISO	SCAMPI	SEI
SPICE		

22-E EXERCISES

22-E.1 Vocabulary

Give a precise definition of each of the entries in the above vocabulary and acronym list.

22-E.2 Stakeholders

Are *competitors* stakeholders in a software project? Discuss what part they, or concerns about them, may play in building the software and managing the project.

Wrong or late?

The overview of CMMI listed under “Further reading” attributes this comment to an unnamed senior manager: “*I’d rather have it wrong than have it late. We can always fix it later*”. Discuss this statement from a software engineering perspective.

23

The software process

24

Writing requirements and documentation

25

Designing Graphical User Interfaces

26

Testing and debugging

27

Towards software reuse

PART VI:

Appendices

W

A

Using the EiffelStudio environment

Throughout this book you have been invited to write examples and run them using the EiffelStudio environment. In the present Appendix you will find all the practical details needed to prepare the examples and set up the execution.

The various sections refer to the successive examples. Be sure to read the first, “EiffelStudio Basics”, before turning to the specifics of any particular example.

A.1 EIFFELSTUDIO BASICS

A.2 EXAMPLE AND EXERCISE SETUP

With EiffelStudio you build *systems*. A system is a collection of classes, grouped for convenience into *clusters*, with one of the classes serving as root, that is to say, the place where execution starts.

← “*SYSTEM EXECUTION*”, 6.8, page 128.

This book contains a number of examples, to which the instructors (if you are taking a course based on TRAFFIC) may have added their own exercises. Rather than setting up each example and exercise as a separate project, we’ve made things more convenient for you by organizing everything into a **single system** called *Traffic*. You will always start that same project, then use one of the buttons on the right — as shown on the following figure — to select an example or exercise. This means that you may:

- Work on different exercises during a single session, without restarting EiffelStudio or opening a new “project” from EiffelStudio.
- Retain the solutions you have written for earlier exercises, and revisit them later on.
- If you develop interesting classes for an exercise, *reuse* them easily for another exercise: they’re all part of the same system and hence automatically available for new developments.

***The example
row***

Example and exercise buttons



If every exercise had its own system, each of these systems would have its own root class and root creation procedure. With a single system, the creation procedure simply calls a different command — corresponding to what would have been an individual creation procedure — depending on the Exercise button clicked.

***Touch of Methodology:
Back up your work!***

It's easy to make a mistake. Whenever your software reaches a stage where it includes new elements that you would hate losing, preserve the results by making a *backup*. A backup is simply a copy of important files. A good backup must:

- Contain *everything* you want to preserve.
- Be stored *separately* from the original data it's intended to preserve, to minimize the likelihood of losing both at the same time.
- *Really* contain what it's supposed to contain; check this periodically, to avoid believing that you have a backup and then — when you need it — realize it's empty or incomplete.

Don't smile at the elementary nature of this advice; many supposedly smart people have been *very* upset at losing important data for failing to observe one of these common-sense rules. It's no fun to program for a week and have to redo everything. On the good side, it takes less time the second time around. Beyond 2, I don't know of any conclusive studies plotting against n the time to redo something you have already done $n - 1$ times.)

Performing repeated backups of an entire software project soon becomes cumbersome. The professional's solution is to use a **configuration management system** that partly automates backups, stores a description of differences between successive versions rather than the full versions, and thanks to this information enables you, if you want to step back, to reconstruct any earlier version of the system.

← "[CONFIGURATION MANAGEMENT](#)",
15.5, page 356.

A.3 SETTING UP THE PROJECT

In the very first example you have to start EiffelStudio on the pre-existing *Traffic* system. The procedure is the following:

← "[A CLASS TEXT](#)",
2.1, page 17.

-
-
-

A.4 BRINGING UP A CONTRACT VIEW

In the study of interfaces you are asked to bring up the contract view of a class, *SIMPLE_LINE*. Bring up the system as described in the previous section. Then click the Exercise button that reads "" and proceed as follows:

← "[What characterizes a metro line](#)", page 57.

27.1 SHOWING A CLASS AND ITS QUERIES

Also as part of the study of interfaces, you have to start the exercise called *Metro* by clicking the corresponding button. You must then locate its class *QUERIES* as follows.

← "[Experimenting with queries](#)", page 59.

-
-

A.5 RUNNING QUERIES ON A SYSTEM

To explore results of queries on the Metro line just built, you may

← End of the section
"[Building a line](#)", page 63.

A.6 SPECIFYING A ROOT CLASS AND CREATION PROCEDURE

For the discussion of creation procedures, you need to specify the root class and root creation procedure of a system. The system already exists and is called *Traffic*; you can see its root class by ← “*SYSTEM EXECUTION*”, page 128.

A.7 CONTROLLING EXECUTION AND INSPECTING OBJECTS

A.8

A.9

A.10 RECOVERING FROM “FUBAR”

(*Fubar* is a computer-folklore acronym whose polite expansion is “Fouled Up Beyond All Repair”.) If you start following the instructions of this chapter, make a mistake, and get to a point where nothing seems to work any more, you may wish to restart from scratch, with the files as they were the first time around.

This is also useful if you are reusing a computer on which someone else has already performed the exercises, using his own solutions.

Warning: if you follow this procedure, everything you have changed or added to the original program texts will be lost!

Here’s how to proceed:

[TO BE COMPLETED.]

Eiffel syntax specification

For reference, the following sections list:

- The keywords of Eiffel.
- All the syntax productions appearing in the text, each with the page of first definition.

The lists do not take into account a few constructs of the language not used in this book. For the complete specification of Eiffel see the official language description: Bertrand Meyer, *Eiffel: The Language*, Prentice Hall.

B.1 KEYWORDS

[To be completed.]

B.2 SYNTAX PRODUCTIONS

$A \triangleq$	<i>Def1 / Def2</i>	336
$A \triangleq$	B C [D] { E ";" ...}* 336	
$A \triangleq$	B Concat 336	
Another_argument \triangleq	" , Identifier	347
Argument_list \triangleq	" (Identifier Another_argument * ")	347
Compound \triangleq	{ Instruction ";" ...}* 333	
Concat \triangleq	C [D] Repet 336	
Conditional \triangleq	if Then_part_list [Else_part] end	334
Else_part \triangleq	else Compound	334
Feature_call \triangleq	Identifier " . Identifier [Argument_list]	347
Repet \triangleq	{ E ";" ...}* 336	
Then_part \triangleq	Boolean_expression then Compound	334
Then_part_list \triangleq	{ Then_part elseif ...} ⁺	334

C

The C# language

C.1

C.2 KEY CONCEPTS LEARNED IN THIS APPENDIX

-

New vocabulary

A

C-E EXERCISES

C-E.1

“”

D

The Java language

D.1

D.2 KEY CONCEPTS LEARNED IN THIS APPENDIX

-

New vocabulary

A

D-E EXERCISES

D-E.1

E

The C language

E.1

E.2 KEY CONCEPTS LEARNED IN THIS APPENDIX

-

New vocabulary

A

E-E EXERCISES

E-E.1

F

The C++ language

F.1

F.2 KEY CONCEPTS LEARNED IN THIS APPENDIX

-

New vocabulary

A

F-E EXERCISES

F-E.1

Picture credits

Page [531](#): Trygve Reenskaug.

Pages : photograph by the author.

Index

Page numbers in **boldface** indicate a page where the concept is defined.

In the electronic version of this index, clicking on a page number will transport you to the corresponding occurrence.

A

abstract complexity **255**
abstract syntax tree **46**
activation record **296, 398**
activations **398**
address **317**
algorithm **141**
allocation
 dynamic **399**
 static **398**
antecedent **86**
API **53**
argument **35**
associative **85**
attached **112**

B

backup **574**
Backus, John **327**
Big-O notation **255**
bit **314**
Black **314**
BNF **327**
BNF-E **329**
Böhm, Corrado **184**
Boolean **66**
break **49**
Brown, Jerry **245**
byte **316**

C

C# **508**
call chain **295**

Chomsky, Noam **349**
class **54**
 generic **245**
 type **247**
client **51**
client programmer **65**
closed hashing **286**
code **43**
coder **43**
collision (in hashing) **285**
Columbus, Christopher **505**
command **33**
comment **19**
 header comment **58**
communication device **9**
commutative **77**
compiler **13, 94**
complexity, abstract **255**
Compound **333**
compound instruction **147**
compound, see compound instruction
computer **5, 8**
concatenation **332**
concurrent **144**
Conditional **334**
conditional **144**
conjunction **78**
Conjunction Principle **78**
consequent **86**
construct **44, 47**
 lexical **343**
container **241**
Contract **71**

contradiction **81**
 control (Windows) 502
 core memory **322**
 correct **13**
 correctness **246**
 CPU (Central Processing Unit) **9**

D

data **10**
 used in the singular 10
 data abstraction 545
 declaration **19, 199, 602**
 defining production **336**
 delimiter **48**
 design **54**
 Dijkstra, Edsger W. 16, 185
 disjunction **77**
 Disjunction Principle 77
 Divide and conquer 198
 Divide and rule 198
 duality **78**
 dynamic allocation **399**
 dynamic property **13**

E

EBNF (extended BNF) 329
 Else_part 334
 embedded **11**
 Enigma 163
 entity **112, 232**
 variable **220**
 Entscheidungsproblem 163, 209
 event
 publishing (same as triggering) 505
 raising (same as triggering) 505
 triggering 505
 exception 93, 115
 Excluded Middle Principle 76, 80
 existential quantifier **98**
 expression **40**
 extendibility **130**
 extendible **13**
 Extreme Cases Principle 258

F

failure 115
 Fairy

Tooth 256
 feature **33**
 feature call **26**
 field **112**
 FIFO (first-In First-Out) 292
 flowchart **183**
 forest **46**
 function
 hash **284**
 Fundamental Data Structure Library Principle 283

G

generating class **54**
 generic
 class **245**
 derivation **245**
 genericity **245**
 glue code 511, 522
 Gödel, Kurt 163
 grammar **44, 328**
 lexical **343**
 GUI **52**

H

hardware **5**
 Hash function **284**
 perfect **285**
 hashing
 closed **286**
 open **285**
 header
 comment **58**
 of a list **278**
 heap **296**
 Heisenberg, Werner 163
 Hertz **319**
 heuristics 340
 Hilbert, David 163

I

I/O (abbreviation for "input and output") **9**
 Identifier
 precise form 48
 identifier **47**
 implementation **54, 199**
 implication **86**
 Implication Principle 86
 indentation 21

information **10**
 information hiding **204**
 input **9**
 instance **54**
 of a routine **397**
Instruction **333**
 instruction **40**
 compound **147**
 interface **51**
 internal node **46**
 International Standards Organization (ISO) **329**
 Invariant Principle **71**
 invariant, see class invariant, loop invariant
 Isis **314**
 item **241**
 iterator **274**

J

Jacopini, Giuseppe **184**
 Jerry **314**

K

keyword **19**
 Knuth, Donald E. **328**

L

language
 natural **41**
 programming, see programming language
 leaf **46**
 Leibniz, Gottfried Wilhelm von **163**
 lexical **48**
 lexical construct **343**
 lexical grammar **343**
 library **129**
 LIFO (Last-In First-Out) **292**
 list
 header **278**
 loop **144**
 invariant **153**
 variant **160**
 Loop Postcondition Principle **189**

M

magic **19**
 manifest string **233**
 memory **8, 12**

 core **322**
 primary **321**
 secondary **322**
 message URL http
 //archive.eiffel.com/products/parse.html **342**
 metalanguage **329**
 methodology advice
 don't box in your users **252**
 standard feature names **253**
 Model-View Separation Principle **513**

N

natural language **41**
 Naur, Peter **328**
 negation **76**
 nested **44**
 nesting **176**
 nesting (of routines) **296**
 Non-Contradiction Principle **76, 80**
 non-strict **94**
 nonterminal **47, 330**
 NYSE (New York Stock Exchange) **509**

O

object **33**
 object-oriented **28**
 octet **316**
 O-O, see object-oriented
 open hashing **285**
 operator **47**
 opposite **76**
 optimization **94**
 optional construct (in a Concatenation production) **332**
 Osiris **314**
 output **9**

P

parallel **144**
 parser **337**
 partial evaluation **484**
 perfect hash **285**
 persistence **9**
 persistent **320**
 phrase **328**
 Polish notation **294**
 Postcondition Principle **70**
 precondition **65**

-
-
- Precondition Principle 67
 - preserve 157
 - primary memory 321
 - Principle
 - Conjunction 78
 - Disjunction 77
 - Excluded Middle 76, 80
 - Extreme Cases 258
 - Fundamental Data Structure Library 283
 - Implication 86
 - Invariant 71
 - Loop Postcondition 189
 - Model-View Separation 513
 - Non-Contradiction 76, 80
 - Postcondition 70
 - Precondition 67
 - Reference Programming 282
 - Symbolic Constant 233
 - Uniform Access 230
 - processor 9
 - production 330
 - defining 336
 - programmer 6
 - programming language 13, 41
 - prototype (C, C++) 497
 - pseudocode 110, 601
 - publishing an event (same as triggering) 505
- Q**
- quantifier 98
 - query 33
- R**
- raising
 - an event (same as triggering) 505
 - RAM (Random Access Memory, synonym for main memory) 321
 - read 315
 - recursion 359
 - recursive 338, 359
 - recursive definition 359
 - Reference Programming Principle 282
 - reflexive 82
 - reusability 130
 - reusable 13, 129
 - robust 13
 - root class 128
 - root creation procedure 128
 - root object 602
 - root procedure 128
 - routine 145
 - nesting 296
 - Russell, Bertrand 163
- S**
- satisfiable 81
 - satisfies 79
 - Schützenberger, Marcel-Paul (1920-1996) 349
 - secondary memory 322
 - semantics 40
 - used as singular 40
 - semistrict 94
 - sequence 144
 - signature
 - function 206
 - software 5
 - source 13
 - special symbol 48
 - specimen 44, 330
 - static allocation 398
 - static property 13
 - storage 322
 - strict 94
 - string
 - manifest 233
 - stronger 87
 - subject (observer pattern), see publisher
 - subprogram 198
 - subroutine 198
 - supplier 51
 - Swing 508
 - Symbolic Constant Principle 233
 - syntax 40
- T**
- target 13
 - tautology 80
 - temporary variable 220
 - terminal 47, 330
 - Then_part 334
 - Then_part_list 334
 - TLA 53
 - token 47
 - Tom 314

Tooth Fairy 256
top construct **331**
transient **320**
tree 45
triggering an event 505
truth assignment **79**
truth table **76**
Turing Award 163
Turing machine 163, 184, 349
Turing, Alan 163, 349
type
 class **247**

U

Uniform Access Principle 230
universal quantifier **98**
user **6**

V

validity **246**
variable **220**
 temporary **220**
variable entity **220**
variable, use "entity" instead **112**
variant, see under loop
Visual Basic .NET 508
vocabulary **328**
void reference **113, 602**

W

water, how to boil 137
weaker **87**
White 314
widget 502
Windows Forms 508
Wirth, Niklaus 185, 328
word **316**
write **315**

Change log

(Will not appear in final version)

- 16.06, 3 December 06: More on software engineering.
- 16.05, 3 December 06: Working on the software engineering chapter.
- 3 December 06: Version 16.04.00 Revised instructor's preface.
- 31 August 06: Version 19.08 Continuing on CMMI
- 30 August 06: Version 19.07 CMMI
- 27 August 06: Version 19.06 Advancing the software engineering chapter.
- 25 August 06: Version 19.05.01 Format change (reduced printing scale to 0.758 to achieve the equivalent of 11pt for basic text and widen the margins).
- 24 August 06: Version 19.05 Advancing the software engineering chapter.
- 23 August 06: Version 19.04 Advancing software engineering.
- 21 August 06: Version 19.03 A little more on software engineering.
- 20 August 06: Version 19.02 Started software engineering chapter.
- 18 August 06: Version 19.01 Some cosmetic changes on recent chapters.
- 17 August 06: Version 19.00 Agent chapter finished.
- 16 August 06: Version 18.60 Agents not quite finished, but definitely tomorrow.
- 15 August 06: Version 18.50 Furthered agent chapter. I hope to finish tomorrow.
- 12 August 06: Version 18.00 Split agent and event chapters. Finished event chapter. Split control structure chapter (new chapter: routines). Split interface chapter (new chapter: program structure). Removed some empty chapters. Made numerous other corrections. (15 July-16 August).
- 2 April 06: Version 16.03.00 Continuing agent-event chapter
- 6 March 06: Version 16.02.02 Continuing agent-event chapter
- 6 March 06: Version 16.02.01 Restarting on agent-event chapter
- 18 February 06: Version 16.00.05 Updating prefaces
- 27 December 05: Version 16.00.05 Corrected a number of details throughout text
- 28 September 05: Version 16.00.06 Restarting with work on event-driven design chapter.
- 16.00.05, 23 November 04: Fixed a few points in topological sort chapter (thanks to comments by Olivier Jeger).
- 16.00.04, 21 November 04: Continuing on assignment
- 16.00.03, 20 November 04: Updating assignment chapter to explain reference assignment and associated issues.
- 16.00.02, 12 November 04: Update to topological sort chapter, on the basis of comments by Olivier Jeger who implemented the solution, on the basis of this chapter, as a new cluster that will be added to EiffelBase.
- 16.00.01, 10 November 04: Small addition to topological sort chapter (where I still have to correct a few errors reported by Olivier Jeger). Most importantly, I have reverted to FrameMaker 5.5.6 (1998!), since 7.1 is hopelessly buggy -- this is sad. Thanks to Ognian Pishev for pointing out that the generated PDF was basically junk.
- 16.00.00, 9 November 04: Finished syntax chapter.
- 15.00.00, 7 November 04: Significant extensions to chapters on data structures (basically finished) and on syntax (added lexical grammars).
- 14.11.01, 13 July 04: Started queues
- 14.11.00, 10 July 04: Finished stacks, can see the end of the data structures chapter.
- 14.10.00, 9 July 04: Working on stack section, about 2/3rds done.
- 14.09.00, 4 July 04: Worked on topological sort and data structures. Added comments about bracket notation, and used it for hash tables in topological sort discussion.
- 14.08.05, 8 May 04: Advancing on linked lists
- 14.08.05, 2 May 04: Working on lists
- 14.08.04, 11 April 04: Finished arrays and hash tables. Working on lists.
- 14.08.03, 10 April 04: Progressing on data structures, arrays etc. chapter
- 14.08.02, 26 March 04: More on arrays etc.
- 14.08.01, 21 March 04: Some advance on data structures
- 14.08.00, 19 March 04: Fixed problems in recursion chapters (a week ago in SB).
- 19.02.01, 9 March 04: Advanced significantly the data structures chapter, now covering genericity as well as algorithm complexity. The Recursion chapter is temporarily in a mess; I'll fix this tomorrow.
- 19.01.01, 9 March 04: Working on data structures chapter
- 19.01.00, 7 March 04: Working on various chapters: recursion, fundamental data structures
- 19.00.01, 2 March 04: Minor updates
- 15.00.00, 1 March 04: The chapter on

- topological sort is ready (with the exception of a couple of figures, “key concepts” etc.) I did perform a spell check but no systematic proofreading. There was intensive rewriting in the past few days, hence the delay. Next is recursion.
- 14.11.00, 26 February 04: Advanced well today. I should definitely have a first full version of the chapter tomorrow.
- 14.10.00, 25 February 04: Advanced topological sort chapter, although I have to stop now to take care of some other matters. Perhaps I will finish tomorrow. Also cleaned up recursion chapter somewhat.
- 14.09.01, 24 February 04: I am now back to where I was two days ago, but the presentation is much cleaner mathematically and pedagogically.
- 14.08.04, 23 February 04: Ended up rewriting most of the existing material (finding out that the mathematical framework that everyone uses to talk about topological sorts, partial orders etc. is just inadequate). I just realized that the result is not right yet, so there will be more work on this. Hence the regression in version numbers.
- [Note added later: I started doing the update, but didn’t finish, so the chapter is in a mess
- 14.09.00, 22 February 04: Finished mathematical background for topological sort. I hope to complete the chapter in one or two days.
- 14.08.03, 21 February 04: Advanced further on topological sort. The next few days should be good.
- 14.08.02, 20 February 04: Advanced just a little.
- 14.08.01, 19 February 04: PLEASE NOTE NEW URL Corrected all typos reported by Philippe Cordel. Working on topological sort chapter
- 14.08.00, 18 February 04: Advanced on topological sort chapter
- 14.07.00, 3 January 04: Again good progress on recursion. Unfortunately there probably won’t be any further updates for a week.
- 14.06.02, 2 January 04: Some more advance on recursion. Didn’t proofread much.
- 14.06.01, 1 January 04: Advanced significantly on recursion, but there is still a lot to do. Tomorrow should be another good writing day.
- 14.05.01, 27 December 03: Continuing on recursion: binary trees
- 14.05.00, 26 December 03: NOTE NEW URL (Santa Barbara) Continuing work on recursion chapter.
- 14.04.02, 24 December 03: Advanced work on recursion chapter
- 14.04.01, 21 December 03: Minor changes
- 14.04.00, 7 December 03: Advanced significantly on assignment, although I still have to write the discussion of dynamic aliasing.
- The text refers to qualified vs unqualified calls, and to Current. Neither notion has actually been introduced. Obviously I need to correct this deficiency.
- 14.03.02, 6 December 03: Continuing on assignment
- 14.03.01, 3 December 03: Intermediate version, not released
- 14.03.00, 30 November 03: Advanced the attributes chapter: covered assignment. I had a lot of trouble figuring out how to explain the concepts, although once I got it it was written in a few hours. I wish there were another week-end day to write about references and dynamic aliasing.
- 14.02.02, 29 November 03: Started on the attributes chapter, but actually spent most of my time today correcting typos in earlier chapters. I have now processed all received comments. Many thanks to those who pointed out errors.
- 14.02.01, 16 November 03: Extended the syntax chapter. There is actually more material than included, but I parked some of it since there was a conceptual problem to which I didn’t have a solution in time for this release.
- 14.02.00, 15 November 03: Reworked the beginning of the syntax chapter, simplifying the presentation and clarifying the terminology (the notions of terminal and token were somewhat mixed up). Updated accordingly section 2.5 (the early intro to language description) in the Objects chapter. All that doesn’t advance the description of assignment and references, but seems necessary.
- 14.01.01, 12 November 03: Started to correct the chapter on syntax
- 14.01.00, 11 November 03: Extended discussion of the halting problem with a historical note, added comments to the presentation of routines, and included a proof of undecidability of the halting problem based on Eiffel.
- 14.00.02, 11 November 03: Fixed a set of typos in the control structure chapter, pointed out (like the one listed next) by Jörg Derungs
- 14.00.01, 11 November 03: Fixed typo in definition of undecidability (6.5)
- 14.00.00, 9 November 03: Essentially finished the control structures chapter; I will return later to the missing sections.
- 13.08.04, 9 November 03: Continuing on routines. Reorganized the chapter to move the notion of algorithm to the beginning.
- 13.08.03, 8 November 03: Working on routines; changed names of classes from LINE to METRO_LINE and STATION to METRO_STATION in conformance with the software.
- 13.08.02, 7 November 03: Corrected a mistake in exercise 4-E.8
- 13.08.01, 4 November 03: Corrected several typos, in particular for “non-strict implication” (3.4, currently page 58, ‘count’ instead of ‘Line8.count’ and a few similar oversights in that chapter; 4.3, currently page 93, methodological rule, mixup between ‘or else’ and ‘and then’; 4.3, currently page 94, comment about ‘implies’, False -> True.)
- 13.08.00, 26 October 03: Significantly advanced the control structures chap-

- ter; a token of self-recognition I am awarding myself a second-level section increase. (Can I then call this an “Award-Winning Book”?) Unfortunately an essential section is still missing: routines and functional abstraction. Once it’s done, the rest of the chapter can wait for a while and I will move on to the next one, on assignment & references.
- 13.07.10, 25 October 03: Discussing gotos and structured programming.
- 13.07.09, 21 October 03: It’s safe to ignore this version; just some progress on slowly completing the control structures chapter.
- 13.07.08, 20 October 03: Changed the first example on request of the TRAFFIC team: Metro.highlight --> Lin8.highlight I do prefer the original, so I hope the change can be undone once the software is brought up to par with the text. Added a few pictures to chapter 1.
- 13.07.07, 19 October 03: Wrote the discussion of algorithms, but not yet of routines (a most important concept of course).
- 13.07.06, 15 October 03: Intermediate; finished conditionals, trying to finish control structures chapter.
- 13.07.05, 14 October 03: Updated EiffelStudio appendix (A) to reflect that there is a single system, rather than one system per example or exercise. Appendix is still very incomplete.
- 13.07.04, 13 October 03: Improved the discussion of constructs and specimens in chapter 2 following remarks by Michela.
- 13.07.03, 10 October 03: Added chapter, empty so far, on program correctness.
- 13.07.02, 9 October 03: Continuing on conditionals. Cleaned up many other elements and changed the style of boxes. I was promised a nice comb to photograph for page 162.
- 13.07.01, 8 October 03: Advanced a bit on conditionals (almost done). Corrected errors in other places.
- 13.07.00, 7 October 03: Working on the description of conditional instructions. Revisions on previous material too.
- 13.06.08, 5 October 03: NOTE NEW URL (shouldn’t change again for a good while). Not much visible progress but cleaned up the discussion of loops. Added two exercises on exclusive or (4-E.9 and 4-E.10, the latter possibly more of a surprise).
- 13.06.06, 23 September 03: Proofread and improved the discussion of loops, fixed many typos.
- I need to drive to Germany and risk my life (or a fine) taking picture at the entrance and exit of a tunnel, preferably in a picturesque setting (see page 29). Any suggestion?
- 13.06.05, 22 September 03: Almost finished loops; worked on the instructor’s preface to integrate changes from the PSI article.
- 13.06.04, 21 September 03: NOTE NEW URL Improved discussion of loops. It now starts with a sketch of a significant example, taken from TRAFFIC. I had previously avoided this to make sure that the notion of invariant appeared at the very beginning; now it comes up a little later but the new organization should be more in line with the rest of the book.
- 13.06.03, 20 September 03: Please note new URL
- 13.06.02, 14 September 03: Continuing on loops (section 6.4)
- 13.06.01, 8 September 03: It seems yesterday’s delivery encountered a problem; I hope this one corrects it.
- 13.06.00, 7 September 03: Added a discussion of the notion of loop termination and loop variant. There won’t be much more until this Friday.
- 13.05.00, 30 August 03: Added to creation chapter a section on correctness.
- 13.04.00, 28 August 03: Traveling last week, but advanced description of control structures; loop section in particular is almost done. Note new URL.
- 13.00.00, 18 August 03: I moved the description of BNF to a separate chapter because it was too long an interlude in the discussion of control structures. The chapter may not be at the right place since it comes in the middle of a discussion of software design issues; I will reassess its proper place later.
- Sorry for the large number of typos in yesterday’s delivery. I hope no one wasted his time correcting them. There are probably still quite a few.
- In the next 10 days I will be working on the text but probably won’t be able to post new versions since I will have no direct Unix access.
- 12.03.01, 18 August 03: Cleaning up the text before leaving for conferences in Japan and Austria tonight.
- 12.03.00, 17 August 03: Almost finished the discussion of syntax description. Somewhere along the way I also added an appendix summarizing the syntax (only those constructs introduced in the book).
- 12.02.02, 15 August 03: Minor progress only, you needn’t bother with this update (continuing on explanations of BNF). I do hope for a productive week-end, though.
- 12.02.01, 13 August 03: Minor extensions to control structures chapter.
- 12.02.00, 10 August 03: A bit more on sequences, also dabbled into recursion.
- 12.01.00, 9 August 03: Wrote and almost finished the section on sequences. I still have to talk about sequence correctness (preconditions and all). Then it will be on to conditionals and loops.
- Sorry for the spurious message from the ETH address yesterday -- a case of fiddling with the files there, faithfully watched by the Cron job.
- 12.00.03, 8 August 03: NOTE NEW URL (at Monash)
- Progressed just a bit on control structures.
- The change log now appears in the text

- itself (at the end, with a link on the copyright page after the title).
 sendmail now works on my Monash machine.
- 12.00.01, 7 August 03: Continuing on control structures, which I am trying to present as problem-solving techniques.
- 12.00.00, 5 August 03: Finished creation chapter (currently 5); probably many errors. On to control structures chapter anyway.
- NOTE: the version indication will say "Zurich" although I am currently in Melbourne. This is because sendmail is broken on my machine here, so I have to run the script through my Sparc at ETH.
- 11.12.01, 27 July 03: This is an interim version, not worth anyone spending his time looking at the details, but ensuring that the record on the site will be up to date. I haven't finished the creation chapter yet although it's slowly taking shape. I spent some time taking into account comments by Karine, which were (I hate to admit it) quite justified. I realized that I need the notions of procedure, function and attribute, but don't know yet where to introduce them.
- 11.12.00, 24 July 03: Progressed on creation chapters but only a little as I found it necessary to add stuff to the logic chapter. It turned out that the part about non-strict operators, added somewhat on a whim yesterday, is really indispensable to the rest of the discussion. Added a few exercises in different places.
- 11.11.00, 23 July 03: Due to a mistake on my part there was no dispatch yesterday.
 Still didn't extend the creation chapter, but added to the Logic chapter the missing section about non-strict operators. Added a Tintin example to the discussion of implication in that chapter. (Permissions will have to be requested.)
 Added some empty chapters: concurrency, event-driven design.
- 11.10.01, 22 July 03: Added an important exercise to the first chapter.
 I was distracted from the groundwork by my promise to provide for this week a paper to the National Japanese O-O conference where I am speaking in August. Actually all I was really asked was an extended abstract, but I decided to use material from the book's preface to produce a stand-alone article on our project (even though at the conference I will be talking about something else, about which a paper has already been published elsewhere). That article is now available at
<http://www.inf.ethz.ch/~meyer/publications/teaching-ispj.pdf>
 In the process of adapting the material, I ended up rewriting it significantly; I have now reflected most of the changes back to the preface, with the exception of the section comparing our approaches with others such as Abelson/Sussman, which I haven't decided about yet.
- 11.10.00, 21 July 03: Continuing on creation chapter. I didn't go as far as I wanted because of various interruptions but the chapter is taking shape -- it's certainly much better organized than yesterday.
- 11.09.00, 20 July 03: - Corrected typos found by Karine Arnout - Added appendix on the use of EifelStudio (currently appendix A). As a result, removed all the placeholders in earlier chapters for explanations on how to start the examples, replacing them by links to sections in that appendix. The sections are still placeholders but at least there are almost no more "To be completed" mentions in the text, and the description of EifelStudio use doesn't pollute the presentation of the concepts. - Continued working on creation chapter (currently chapter 5), which actually introduces many other important concepts besides creation: system execution, root class, root creation procedure, entity.
- 11.06.01, 18 July 03: Started working on creation chapter. Also, I just realized I was sending the update messages to the `traffic` list rather than to `touch`; if you are just on the latter you didn't receive recent versions. Sorry for the confusion.
- 12.00.00, 16 July 03: Mostly finished the Interface chapter. Please note that the URL of the current version is different.
- 11.05.04, 15 July 03: Typo correction
- 11.05.03, 15 July 03: During London-LA flight, improved Instructor's preface, corrected many typos, started description of invariants in Interface chapter.
- 11.05.02, 6 July 2003: Prepared overall book structure Started writing Introduction for the instructor (moved it ahead of introduction for the student).