



# The Evolution of Oberon-2 to Component Pascal

Except for some minor points, Component Pascal is a superset of Oberon-2. Compared to Oberon-2, it provides several clarifications and improvements. This text summarizes the differences. Some of the changes had already been realized in earlier releases of the BlackBox Component Builder, all of them are implemented for Release 1.3.

The language revision was driven by the experience with the BlackBox Component Framework, and the desire to further improve support for the specification, documentation, development, maintenance, and refactoring of component frameworks. The goal was to give a framework architect the means to better control the overall integrity of large component-based software systems. Control over a system's integrity is key to increased reliability, reduced maintenance costs, and to higher confidence in the system's correctness when it evolves over time.

Care was taken that the language remains small, easy to use, and easy to learn. The new features are most visible to framework designers, less visible to framework extenders, and least visible to mere framework clients. This ensures that these different categories of developers are burdened with the minimal amounts of complexity that their respective tasks require. The complete language report is available [here](#).

## More Expressive Type System

### Covariant pointer function results

A type-bound function which returns a pointer may be redefined, such that it returns an extended type. For example, the function

```
PROCEDURE (v: View) ThisModel (): Model
```

could be extended in a subtype MyView, which is assumed to be a subtype of View, to the following function signature

```
PROCEDURE (v: MyView) ThisModel (): MyModel
```

where MyModel is assumed to be a subtype of Model. Note that covariant function results are type safe; they simply strengthen the postcondition of a function, which is always legal. They allow to make interface declarations more precise.

### Pointer compatibility

The compatibility rules for pointers have been relaxed and simplified. Pointers are now compatible by structure; i.e., two pointer types that have the same base type are compatible. This can be useful mostly in procedure signatures, where it previously wasn't possible to use a function like the following:

```
PROCEDURE P (p: POINTER TO ARRAY OF INTEGER)
```

### Pointer function results

A function's return type is now Type not Ident; e.g.,

```
PROCEDURE Bla () : POINTER TO Rec
```

is now legal, and due to simplified compatibility rules (see previous point) their use can actually make sense.

## IN and OUT

These parameter modes are like the VAR mode, except that some restrictions apply. IN parameters cannot directly be modified inside a procedure, OUT parameters are considered undefined upon procedure entry (except for pointers and procedure variables, which are set to NIL upon procedure entry). OUT record parameters must have identical actual and formal parameter types.

These parameter modes are important for procedure signatures of distributed objects, and they can increase convenience and efficiency. Most importantly, they allow to make interface declarations more precise and more self-documenting. In particular, where formerly VAR parameters have been used for efficiency reasons only, it is now possible to use IN parameters. Where efficiency is not critical, normal value parameters should be used instead of IN parameters.

Example:

```
PROCEDURE ShowModes (
  value: INTEGER; VAR inout: INTEGER; (* value and VAR parameters *)
  IN in: ARRAY OF SET; (* IN for efficiency *)
  OUT res: INTEGER): INTEGER; (* OUT parameter and function result *)
```

## NEW methods

Component Pascal requires that the introduction of a new method is indicated explicitly. This is done by appending the identifier NEW to the method's signature. NEW may not be used for extending methods.

In the following example, method SomeMethod is newly introduced in T and inherited in T1, which is assumed to be an extension of T:

```
PROCEDURE (t: T) SomeMethod (x, y: INTEGER), NEW;
BEGIN
  ...
END SomeMethod;

PROCEDURE (t: T1) SomeMethod (x, y: INTEGER);
BEGIN
  ...
END SomeMethod;
```

NEW indicates that a method is new, not extending. The need to declare this fact explicitly is useful whenever changes to a framework are made, which often happens during the initial design iterations, and later when the software

architecture undergoes refactoring. NEW makes it possible for the compiler to detect for example if a base method's name has been changed, but extending methods have not been renamed accordingly. Also, the compiler detects if a method is newly introduced, although it already exists in a base type or in a subtype. These checks make it easier to achieve consistency again after a change to a framework's interfaces.

## Default, EXTENSIBLE, ABSTRACT, and LIMITED record types

Component Pascal uses a single construct to denote both interfaces of objects and their implementations: record types. This unification allows to freeze some implementation aspects of an interface while leaving others open. This flexibility is often desirable in complex frameworks. But it is important to communicate such architectural decisions as precisely as possible, since they may affect a large number of clients.

For this reason, a Component Pascal record type can be attributed to allow an interface designer to formulate several fundamental architectural decisions explicitly. This has the advantage that the compiler can help to verify compliance with these decisions. The carefully chosen attributes are EXTENSIBLE, ABSTRACT, and LIMITED. They allow to distinguish four different combinations of extension and allocation possibilities:

<i>modifier</i>	<i>extension</i>	<i>allocation</i>	<i>record assignment</i>
none ("final")	no	yes	yes
EXTENSIBLE	yes	yes	no
ABSTRACT	yes	no	no
LIMITED	no (1)	no (1)	no

(1) except in the defining module

Record types may either be extensible or non-extensible ("final"). By default, or if marked as LIMITED, a record type is final. Final types allow to "close" a type, such that an implementor can perform a complete analysis of the type's implementation, e.g. to find out how it could be improved without breaking clients.

Record types may either be allocatable (as static or dynamic variables), or allocation may be prevented (ABSTRACT) or limited to the defining module (LIMITED).

Final types typically are simple auxiliary data types, e.g.:

```
Point = RECORD
    x, y: INTEGER
END
```

Variables of such types can be copied using the assignment operator, e.g. `pt := pt2`. The compiler never needs to generate the hidden type guard that is sometimes necessary for such an assignment in Oberon.

On the other hand, extensible records can neither be copied, nor passed as value parameters (since value parameters imply a record assignment).

Final types, like extensible types, may be extensions of other record types and they may have methods.

Extensible types are declared in the following way:

```

Frame = EXTENSIBLE RECORD
      l-, t-, r-, b-: INTEGER
END

```

Plain EXTENSIBLE types are rare, it is more typical to use ABSTRACT types instead, which are a special case of extensible types that cannot be instantiated. The following paragraph gives a more precise description of what this restriction means:

Types of values can never be abstract, but types of variables may be abstract. The type of a value can be different from the type of its holding variable only if the variable is referential; i.e., a pointer; or a VAR, IN, or OUT parameter. Thus, only those variables may be declared to be of an abstract type. In all other cases; i.e., static variables, record fields, array base types, and value parameters, non-abstract types or pointers (possibly to abstract types) must be used. Since the allocation operation NEW produces a value of the argument's type, NEW can only be used with variables of non-abstract type.

Example of an abstract type:

```

TextView = POINTER TO ABSTRACT RECORD (Views.View) END

```

Abstract types are design tools, they denote interfaces of objects. They are the primary means of Component Pascal to model component interfaces. Denoting records as abstract allows to indicate more precisely the use of a record: as an interfacing construct rather than an implementation construct.

Nevertheless, an abstract type may have all types of methods (see below), i.e., it is not forced to be fully abstract.

LIMITED types are a special case of final types. They are special in that they can be instantiated only within the defining module, and they cannot be copied. For example, a client may not perform a NEW on variables of limited types. Since allocation is under complete control of the defining module, the programmer of this module can guarantee that all newly allocated variables are correctly initialized before they are made accessible to client modules. This means that clients can only see variables that respect the type's invariants (which are established during initialization). An implementor is free to change the type's internal representation with less risk of breaking client code; there is no need for lazy initialization schemes; there cannot be delayed run-time errors due to missing initializations; and invariants (e.g., invariants over hidden record fields) cannot be violated through copying.

Typically, factory functions or factory objects are provided to create new instances of dynamic LIMITED types.

Example:

```

TYPE
  Semaphore = POINTER TO LIMITED RECORD END;

PROCEDURE New (level: INTEGER): Semaphore;

```

In the BlackBox Component Framework, most abstractions are represented as abstract types, which are implemented by (non-exported) final types. This is another approach that allows to guarantee correct initialization, but it is too inconvenient for simple non-extensible abstractions. Moreover, LIMITED types cannot be substituted by client-side extensions. This is important, because it allows to protect non-extensible services, such as a real-time kernel, from being used with illegal types.

## Record syntax

The record syntax looks as follows:

```

RecordType = [EXTENSIBLE | ABSTRACT | LIMITED]
             RECORD ["(" QualIdent ")"]

```

```
FieldList {";" FieldList} END.
```

Note that a pure client programmer never needs to write any of the above attributes. The same is true for an implementor of a framework extension. Even the framework designer may save some time using these attributes, because they are an important part of the documentation that can be extracted automatically from the source code.

The goal for the introduction of these attributes was to increase the static expressiveness of interfaces, such that important architectural decisions can be written down explicitly, in a way that a compiler can check conformance of an implementation or a client with the interface contract. A pure implementation language wouldn't need the new attributes, only a component-oriented implementation and design language needs to be able to express such design constraints. Control over such constraints enables a framework designer to establish important invariants over a whole software system (= system architecture), thus improving safety, maintainability, and evolvability.

Some of the new attributes also add convenience; e.g., abstract methods need no procedure body anymore. Such additional convenience is a welcome benefit, but it was by no means the reason for the introduction of the attributes.

## Default and EXTENSIBLE methods

Like record types, methods of a record type can also be attributed. The attributes available are the default (no attribute), EXTENSIBLE, ABSTRACT and EMPTY.

Like record types, methods are final by default:

```
PROCEDURE (t: T) StaticProcedure (x, y: INTEGER), NEW;
BEGIN
    ...
END StaticProcedure;
```

Methods that are both new and final can be treated by a compiler like normal procedures, since they don't require late binding. Nevertheless, their use can be appropriate if they clearly belong to a particular type.

Extensible methods on the other hand are marked as such, e.g.:

```
PROCEDURE (t: T) Method (x, y: INTEGER), NEW, EXTENSIBLE;
BEGIN
    ...
END Method;
```

It is much more common to use abstract or empty methods, which are special cases of extensible methods (see below).

Declaring a method as final is achieved by simply leaving away the EXTENSIBLE attribute:

```
PROCEDURE (t: T) FinalInheritedMethod (x, y: INTEGER);
BEGIN
    ...
END FinalInheritedMethod;
```

If a black-box design style is used, most methods that need to be implemented for a framework extension are of the above kind, which requires no special attributes in the method signature. This is reasonable, because there are more framework extension programmers than there are framework designers, thus extensions should be as convenient as possible to write down.

Final methods may be bound to any record types. Extensible methods may only be bound to extensible types (i.e., EXTENSIBLE or ABSTRACT).

Since final methods cannot be "overridden", the invariants that they guarantee and the postconditions that they establish cannot be violated. Note that correct "extension" of a method means that the extending method implements a refinement of the extended method. Semantically, this means that the extending method accepts a weaker precondition or establishes a stronger postcondition compared to the extended method.

## ABSTRACT methods

An abstract method is declared in the following way:

```
PROCEDURE (t: T) SomeMethod (s: SET), NEW, ABSTRACT;
PROCEDURE (t: T) CovariantMethod2 (): NarrowedType, ABSTRACT;
```

Abstract methods are extensible. The compiler checks that a concrete type implements all abstract methods that it inherits. A concrete extension of an abstract method (or type) can be regarded as its implementation. Abstract methods may only be bound to abstract types, and they may not be called via super calls.

An abstract method has no corresponding procedure body, it only exists as a signature. There is no need anymore to write a procedure body with a HALT statement.

## EMPTY methods

A method can be declared as empty. Empty methods are extensible. An empty method is very similar to an abstract method, in that it is a hook for functionality that can be provided in later extensions. However, empty methods are concrete and can be called. If they have not been extended (i.e., implemented), calling them has no effect.

Empty methods represent optional interfaces. For example, a BlackBox Component Framework View provides an empty method for handling user events (HandleCtrlMsg). This method is implemented in interactive views, passive views ignore it.

It is not possible to introduce code in an empty procedure. For this reason, an empty method has no corresponding procedure body, and may not be called via super calls.

Empty procedures may not return function results and may not have OUT parameters.

Example:

```
PROCEDURE (t: T) Broadcast (msg: Message), NEW, EMPTY;
```

### Method syntax

The method syntax looks as follows:

```
TBProc = PROCEDURE Receiver IdentDef [FormalPars] [Attribution].
Attribution = [", " NEW] [", " (EXTENSIBLE | ABSTRACT | EMPTY)].
```

Note that a pure client programmer never needs to write any of these attributes. The same is true for an implementor of a framework extension. Even the framework designer may save some time using these attributes, because they are an

important part of the documentation that can be extracted automatically from the source code.

## Implement-only export of methods

A method may now also be exported as implement-only, using the "-" export mark instead of the "\*". Implement-only export means that the method may be implemented outside the defining module, but may not be called from there.

Whether a method is exported normally or implement-only is decided when the method is first introduced (NEW method). Later extensions must use the same export mode.

Implement-only exported methods are called from within the module where a method is newly introduced; they go "upwards" in the module hierarchy (upcalls). For frameworks, the existence of such upcalls is typical. Implement-only export allows to prevent framework clients from violating the framework's invariants, while still making it possible to provide new implementations of the framework's types.

Every framework has two "faces": an interface for clients, and an interface for implementors, the so-called specialization interface. These two interfaces may overlap. Implement-only export allows to clearly label those parts of an interface that belong to the specialization interface only.

## Super calls

Because of the so-called semantic fragile base class problem, it is recommended to avoid super calls whenever possible. It is possible to design new software such that they are not needed, by relying on composition rather than on implementation inheritance. Super calls are considered to be an obsolete feature. For the time being, they are retained for backward compatibility. In the long run, support for super calls may be reduced.

## Procedure types

Procedure types are less flexible than objects with methods. Even standard examples for procedure types in numerical software can benefit from modeling them as objects. Objects are extensible, procedure types are not. Procedure types can pose considerable implementation difficulties concerning the safe unloading of code. For these reasons, procedure types are considered as obsolete. For the time being, they are retained for backward compatibility and for implementation low-level interfacing code (callbacks). In the long run, support for super calls may be reduced.

## ANYREC and ANYPTR

Each base record is implicitly regarded as an extension of the new abstract standard type ANYREC, even if it is declared without explicit base type. ANYREC is an empty record that forms the root of all record type hierarchies. ANYPTR is a new standard type that corresponds to a POINTER TO ANYREC.

These new types make it easier to achieve interoperability between independently developed frameworks, by allowing completely generic parameters.

The following pseudo definitions can be assumed:

```

ANYREC = ABSTRACT RECORD END;
ANYPTR = POINTER TO ANYREC;
PROCEDURE (a: ANYPTR) FINALIZE-, NEW, EMPTY;

```

The FINALIZE procedure is empty. It can be implemented for a pointer type extension. The procedure is called at some unspecified time after the object has become unreachable via other pointers (not globally anchored anymore) and before the object is deallocated. Finalizers are needed to release resources that are not directly Component Pascal objects; e.g., file sectors, font handles, window pointers of the operating system, and so on.

The finalization order is not defined. An object is only finalized once.

## String Support

### Explicit string types

We can distinguish a string value (the actual character values) from the variable in which it is contained (an array of character). Some operations in Component Pascal operate on the string value (e.g., comparison for equality), others operate on the container variable (e.g., assignment). A compiler can automatically derive which interpretation is needed in a given situation. Unfortunately, there are situations where both interpretations make sense. For example, passing an array of character to a value parameter (which is also an array of character) should be interpreted as an assignment. But often it is more efficient only to copy the string value in the actual parameter, rather than the whole array. Consider passing a Unix path name, declared as an array of character with 2048 elements, when it usually contains only a few dozen characters.

In Component Pascal, it can be made explicit that the programmer wants to work with the string value, rather than the character array variable. Selecting the string value in a variable is denoted with the \$ operator; for example

```
OpenFile (pathname$)
```

where OpenFile is declared as

```
PROCEDURE OpenFile (name: ARRAY 2048 CHAR)
```

Note an additional benefit: it is now more attractive to declare character array parameters with a fixed number of elements, like in the above example. In Oberon, this is inconvenient since often the type of the actual parameter is not compatible with the formal parameter. Since string values are always compatible with character arrays, this problem vanishes in Component Pascal. This is important because it is more precise to specify a fixed size array when the array is known to be limited. Note that declaring an open array would be a contract to accept arrays of any length whatsoever (without ever leading to an out-of-range error at run-time!).

### String concatenation

The + operator now allows to concatenate strings. The target variable must be of sufficient length to hold the resulting string.

### Elimination of COPY

The auxiliary procedure COPY is not necessary anymore, since the \$ operator makes it superfluous. For example,

```
COPY (a, varpar)
```

is replaced by



```
varpar := a$
```

## Specified Domains of Types

To achieve fully portable code, it is necessary to fully specify the domains of all base types. The Component Pascal base types are a superset of the Java base types.

<i>Type</i>	<i>Size</i>	<i>Domain</i>
SHORTCHAR	1 byte	Latin-1 character set (first Unicode page, and a superset of ASCII)
CHAR	2 byte	Unicode
BYTE	1 byte	signed integer
SHORTINT	2 byte	signed integer
INTEGER	4 byte	signed integer
LONGINT	8 byte	signed integer
SHORTREAL	32 bit	IEEE
REAL	64 bit	IEEE
SET	4 byte	bitset
BOOLEAN	1 byte	FALSE or TRUE

Type LONGREAL has been eliminated. Longreal literals have been eliminated; i.e., use 1.0E2 instead of 1.0D2. The identifier LONGREAL is reserved for possible future use. Real constants are always REAL (64 bit) values.

Hexadecimal integer constants now can be specified either as 4 byte (e.g., 0FFFFFFFH) or 8 byte constants (e.g., 0FFFFFFFFFFFFFFFFL). This allows to distinguish negative INTEGER hex constants from positive LONGINT hex constants. For example, 0FFFFFFF denotes -1 when interpreted as INTEGER, but 4294967295 when interpreted as a LONGINT.

Integer constants are always INTEGER (4 byte) values. Assignment of an integer constant to a smaller type (e.g., BYTE) is legal if the constant lies within the range of the target type. Integer constants of other types can only be constructed using SHORT or LONG.

Integer arithmetic is now always performed with 32-bit precision, except for expressions that contain LONGINT values. In the latter case, 64-bit precision is used. This rule makes it less likely to produce hard-to-find overflows of intermediate results that are calculated at insufficient precision.

Floating-point arithmetic is always performed at 64-bit precision.

## Miscellaneous

The semantics of DIV is strengthened, by specifying the result of divisions by negative numbers.

The new real value INF for infinity has been introduced. This value may be generated e.g. through floating-point

division by zero. The meaning of infinity for floating-point numbers is defined by the IEEE standard for floating-point numbers.

It has been specified more comprehensively where pointer dereferencing is implicit. For example, it is now also available when passing a pointer variable to a record type formal parameter.

Record types can be declared as extensions of other record types by mentioning a pointer type as base type (instead of a record type). This makes the explicit naming of a record type superfluous, if the record variables are used via pointers only.

Within a scope, type names can be considered as forward-declared. This means that any type name can be used before it is declared. Old-style pointer forward declarations like `T = POINTER TO TDesc` are still allowed, since they are simply a special case of the new rule, but they are not necessary anymore.

`BITS` is a new standard function that converts an `INTEGER` value to a `SET` value, such that `BITS(1)` yields `{0}`. For example, this allows to write more portable device drivers, since it doesn't depend on the processor's bit ordering (which differs for 68k and PowerPC, for example).

`ORD` can now also be applied to `SET` values (inverse operation of `BITS`).

`MIN` and `MAX` now also accept two parameters; e.g. `real0 := MAX(someInt, real1)`. They select the minimal/maximal values of the two inputs, which must be number types.

`LEN` can also be used on string values. `LEN(chararray)` returns the length of the chararray variable, while `LEN(chararray$)` returns the length of the string value. Note that a character array must have at least one more element than the string value contains characters, to hold the terminating `0X` character.

Designators are generalized to allow dereferencing etc. on a function result. For example, the following is now legal:

```
length := view.ThisModel() (TextModels.Model).Length()
```

or

```
view.Context().GetSize(w, h)
```

The relaxation of the designator syntax makes it easier to use methods instead of record fields. Methods are more flexible and make it simpler to implement wrappers (forwarding of method calls) than record fields do.

Global variables, including heap variables allocated with `NEW`, now have a defined initial value (`FALSE`, `0X`, `0`, `0.0`, `{}`, `NIL`, `""`). Local (i.e., stack) variables are not initialized, except for pointers and procedure variables which are set to `NIL` for safety reasons.

An appendix of the language report specifies the minimal environment requirements that any Component Pascal implementation must fulfill. In particular, commands, dynamic loading, and garbage collection are fundamental requirements for component-oriented systems. Like for all dynamic languages, this appendix acknowledges that the language cannot be regarded completely independently from its environment. The object model, whether assumed in the language definition or accessed as an external service, is always part of the environment.

To simplify interfacing of existing C libraries, underscores in identifiers are allowed.

The rules for export marks have been simplified compared to Oberon: An extending method must have exactly the same export mark as the method that it extends. The only exception occurs if the method is part of a non-exported record; in this case it may not be exported.

A module now has an optional CLOSE section, after the BEGIN section. The close section is called before a module is unloaded. A module's BEGIN section is called after all the imported modules' BEGIN sections have been called. A module's CLOSE section may only be called after all the importing modules' CLOSE sections have been called.

In summary, you'll note that this revised language definition contains one major extension: a more expressive interface definition language (IDL) subset (NEW; EXTENSIBLE; ABSTRACT; LIMITED; EMPTY; implement-only export of methods) that makes it easier to specify architectural properties of a component framework.

The other points are mostly detail improvements based on a decade of experience in using the language. Particularly noteworthy are the more general and systematic treatment of strings and the new specification of the base type sizes.

## Migration

In order to simplify migration from Oberon, Release 1.3 of the BlackBox Component Builder contains a tool that helps in converting old source code. The tool is not a simple textual pattern matcher; it uses several modules of the compiler to guarantee a precise transformation of the code. In particular, type lifting (e.g., LONGINT -> INTEGER), the conversion of mapper procedures (e.g., textwriter.WriteLInt -> textwriter.WriteInt), elimination of superfluous LONGs, and the elimination of COPY are supported. Furthermore, the tool eliminates calls to the now redundant procedures Strings.Concat, Strings.Append and Strings.Len. It correctly distinguishes between normal modules and interfacing modules, and it allows to set the color and font style in which the changed source code stretches are marked.

## Acknowledgements

I would like to thank Beat Heeb, Dominik Gruntz, Matthias Hausner, and Daniel Diez of Oberon microsystems for their valuable input. In particular, without Beat's work this endeavour would have been impossible. The most important external contributions came from Clemens Szyperski (Queensland University of Technology, Australia) and Wolfgang Weck (Åbo Akademi, Finland). Last but not least, I would like to thank Prof. Niklaus Wirth for commenting an early revision of the Component Pascal language report, and for having created such a great foundation.

*Cuno Pfister, Oberon microsystems, Inc.*

*Copyright © by Oberon microsystems, Inc. All rights reserved. No part of this text may be reproduced in any form or by any means, without prior written permission by Oberon microsystems.*

---

Oberon microsystems, Inc.  
Technoparkstrasse 1  
8005 Zürich  
Switzerland

Tel (+41 1 ) 445 1751  
Fax (+41 1) 445 1752  
Net [oberon@oberon.ch](mailto:oberon@oberon.ch)

Back to our [Home Page](#).