Inheritance (cont.)

CS 412/512
Old Dominion University
Steven J. Zeil

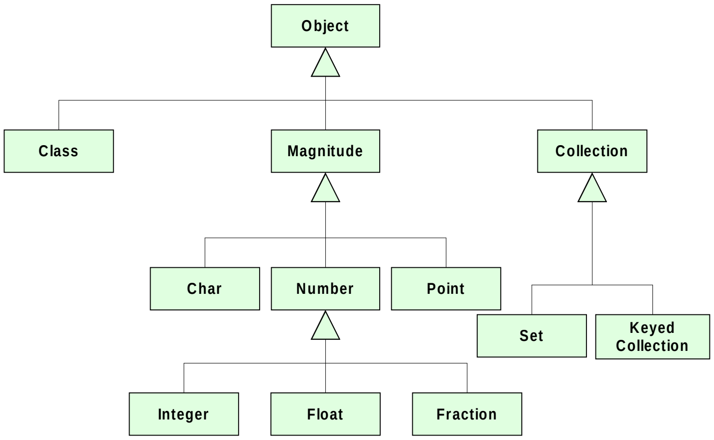
Oct. 4, 2000

Inheritance in Other Languages)

1. Inheritance in Smalltalk
2. Inheritance in Java

1 Inheritance in Smalltalk

- interpreted (usually)
- Everything is an object. (Even classes are objects!)
- All function calls are resolved dynamically.
- All classes are arranged into a single inheritance tree:



```
AShape subclass: #Rectangle
  instanceVariableNames:
   'theHeight theWidth'
  classVariableNames:

setHeight: anInteger
  "set the height of a rectangle"
  theHeight +anInteger

setWidth: anInteger
  "set the width of a rectangle"
  theWidth +anInteger
```

```
height
  "return the height of a rectangle"
  ^theHeight

width
  "return the width of a rectangle"
  ^theWidth
```

```
draw
  "draw the rectangle"
  | aLine upperLeftCorner |
  aLine + Line new.
  upperLeftCorner +
theCenter x - (theWidth / 2)
    @ (theCenter y - (theHeight / 2)).
```

ARectangle inherits a data member theCenter from AShape. theCenter x means “send the “x” message to the theCenter object”.

2 Inheritance in Java

Another hybrid, but purer than C++.

- Not all types are classes.
- All class member functions are dynamically bound.
- All classes organized into a single inheritance tree

2.1 Using Superclasses in Java

Java implementation of Shapes is very similar to C++:

```
class Point {
  double x, y;
}
```

Since this declaration does not explicitly state a superclass, by default it inherits from `Object`.

Object

`Object` is by no means a trivial class. Messages are:

- `protected native Object clone()`

  Creates a new object of the same class as this object.
- `public boolean equals(Object)`

  Compares two Objects for equality.
- `finalize()`

  Called by the garbage collector on an object when there are no more references to the object.

- `public final Class getClass()`

  Returns the runtime class of an object.
- `public native int hashCode()`

  Returns a hash code value for the object.
- `public final native void notify()`

  Wakes up a single thread that is waiting on this object's monitor.
- `public final native notifyAll()`

  Wakes up all threads that are waiting on this object's monitor.

- `public String toString()`

  Returns a string representation of the object.

-
  ```
  public final native wait()
  public final native wait(long)
  public final native wait(long, int)
  ```

  Waits to be notified by another thread of a change in this object.

```
import Point;

class RectangularArea
{
private Point ul;
private Point lr;

  RectangularArea (Point upperLeft,
                   Point lowerRight)
  {
    ul = new Point(upperLeft);
    lr = new Point(lowerRight);
  }

  Point upperLeft()  {return new Point(ul);}
  Point upperRight() {return new Point(lr.x, ul.y);}
  Point lowerLeft()  {return new Point(ul.x, lr.y);}
  Point lowerRight() {return new Point(lr);}

  int width()   {return lr.x - ul.x;}
  int height()  {return ul.y - lr.y;}

  boolean isEmpty() {...}
  static RectangularArea empty() {...}

  boolean contains (Point p) {...}

  boolean overlaps ( RectangularArea r) {...}

  void merge ( RectangularArea r) {...}
}
```

As in C++, we often define abstract classes to establish a common protocol:

```
abstract
class Shape {
  abstract void draw();
  abstract void zoom (Point origin,
                      double factor);
  abstract Point center();
  abstract RectangularArea bound();
}
```

Establishes the common interface for all shapes.

```
class ShapeList {
   Shape shape;
   ShapeList next;
};
```

Since all class objects are assigned by reference, no need for explicit pointers.

```
class Picture {
   private ShapeList shapes;

   Picture()  {...}

   void clear() {...}

   void add (Shape s) {
     ShapeList newNode = new ShapeList;
     newNode.shape = s;
     newNode.next = shapes;
     shapes = newNode;
     }

   RectangularArea bound() {...}
   void draw() {...}

   void zoom (Point origin,
              double factor) {...}
};
```

```
class Circle extends Shape {
   private Point theCenter;
   private double theRadius;

   Circle (Point cent, double r) {...}

   void draw() {...}
   void zoom (double factor) {...}
   RectangularArea bound() {...}
   double radius () {return theRadius;}
}
```

Drawing a picture:

```
class Picture {
  ⋮
  void draw() {
  {
    ShapeList s = shapes;
    while (s != null) {
      s.shape.draw();
      s = s.next;
     }
   }
}
```

2.2 Interfaces In Java

Java offers an alternate mechanism for subtyping, the interface. An interface declares a related set of

- member function declarations
- constant values

Classes may be declared to implement an interface independently of where they are in the inheritance hierarchy.

Example: you would like to write a sorting routine for Java.

- Now all sorting algorithms require the ability to compare objects.
- But class `Object` has no comparison function except `equalTo`.

One solution is to define the “comparable” protocol as a class.

```
class Comparable {
  public boolean comesBefore (Object o)
    {return hashCode() < o.hashCode();}
}
```

(Not a very useful default.)

20

```
class Sorting {

  public static void
    insertionSort (Comparable[] array,
                   int nElements)
  {
    for (int i = 1; i < nElements; ++i) {
       Object temp = array[i];
       int p = i;
       while ((i > 0)
              && temp.comesBefore(array[p-1])) {
          array[p] = array[p-1];
          p--;
       }
       array[p] = (Comparable)temp;
    }
  }
    ⋮
}
```

‡

21

```
class Student
  extends Comparable
{
  String name;
  String id;
  double gpa;
  String school;

  boolean comesBefore(Object o)
  {
    return gpa > ((Student)o).gpa;
  }

}
```

22

Two big problems with this approach:

- What if, in the same program, we want to sort students by `name`?
- What if `Student` is already inheriting from another class?

23

```
class Person
{
  String name;
  String id;
}

class Student
  extends Person
{
  double gpa;
  String school;
}
```

Java only allows a class to have a single superclass, so we can’t add `extends Comparable` ‡

24

2.2.1 Interfaces

One solution is to use an interface:

```
interface Comparable {
  boolean comesBefore (Object o);
}
```

25

The code for the sort does not change:

- interfaces are types

```
public static void
  insertionSort ( Comparable [] array,
                  int nElements)
{
  for (int i = 1; i < nElements; ++i) {
     Object temp = array[i];
     int p = i;
     while ((p > 0)
            && temp.comesBefore(array[p-1])) {
        array[p] = array[p-1];
        p--;
     }
     array[p] = temp;
  }
}
```

26

Now we indicate that `Student` *implements* the interface:

```
class Student
  extends Person
  implements Comparable
{
  double gpa;
  String school;

  boolean comesBefore(Object o)
  {
    return gpa > ((Student)o).gpa;
  }

}
```

Implementing an interface is *NOT* inheritance:

- You cannot inherit variables from an interface.
- You cannot inherit method implementations (function bodies) from an interface.
- The interface hierarchy is independent of a the class hierarchy.
- A Java class may implement many different interfaces, but can only inherit from one superclass.

Of course, interfaces can be more complex than `Comparable`:

```
interface Collection {
  int MAXIMUM = 500;

  void add(Object obj);
  void delete(Object obj);
  Object find(Object obj);
  int currentCount();
}
```

```
class FIFOQueue
  implements Collection {
    ...
    void add(Object obj) {
        ...
    }
    void delete(Object obj) {
        ...
    }
    Object find(Object obj) {
        ...
    }
    int currentCount() {
        ...
    }
}
```

```
package java.util;

/**
 * An object that implements the Enumeration interface generates a

 * series of elements, one at a time. Successive calls to the
 * nextElement method return successive elements of the
 * series.
 *
 * For example, to print all elements of a vector <i>v</i>:
 *
 * for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
 * System.out.println(e.nextElement());
 * }
 *
 */
```

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration.
     */
    Object nextElement();
}
```

```
public interface AudioClip {
    /**
     * Starts playing this audio clip. Each time this method is called,

     * the clip is restarted from the beginning.
     */
    void play();

    /**
     * Starts playing this audio clip in a loop.
     */
    void loop();

    /**
     * Stops playing this audio clip.
     */
    void stop();
}
```

2.2.2 Providing Multiple Methods

Remember that we identified two big problems in the sorting program:

- What if, in the same program, we want to sort students by `name`?
- What if `Student` is already inheriting from another class?

Interfaces help the second, but not the first.

Possible solutions:

1. Special-case subclasses
2. Special-case subclasses with Indirection
3. Functors

Special-case subclasses

We could make a special subclass of student that overrides the `comesBefore` method:

```
class StudentsByName
  extends Student {

  StudentsByName (Student s) {
   super(s); // invoke superclass's constructor
  }

  boolean comesBefore (Object o) {
    StudentsByName s
       = (StudentsByName)o;
    return name.compareTo(s.name) < 0;
  }
}
```

So to sort a group of students by name, we must first copy them to/from an array of `StudentsByName`.

```
void sortByName (Student[] sarray,
                 int nElements) {
  StudentsByName[] tempArray
      = new StudentsByName[nElements];

  for (int i = 0; i < nElements; ++i) {
    tempArray[i] = new
      StudentsByName(sarray[i]);
  }

  Sorting.insertionSort (tempArray, nElements);

  for (int i = 0; i < nElements; ++i) {
    sarray[i] = new Student(tempArray[i]);
  }
}
```

Special-case classes with Indirection

A somewhat cheaper solution (even more so in a language with copy semantics) is to introduce a level of indirection.

> Koenig's fundamental theorem of software engineering:
> "We can solve any problem by introducing an extra level of indirection."

‡

```
class StudentsByName {

  Student st;

  boolean comesBefore (Object o)
  {
    StudentsByName s
       = (StudentsByName)o;
    return st.name.compareTo(s.st.name) < 0;
  }
}
```

39

```
void sortByName (Student[] sarray, int nElements) {

  StudentsByName[] tempArray
     = new StudentsByName[nElements];

  for (int i = 0; i < nElements; ++i) {
    tempArray[i] = new StudentsByName;
    tempArray[i].st = sarray[i];
  }

  Sorting.insertionSort (tempArray, nElements);

  for (int i = 0; i < nElements; ++i) {
    sarray[i] = tempArray[i].st;
  }
}
```

This is cheaper because only *references* to the `Student` objects are copied.

40

Functors

A still more elegant solution is to redesign the sorting function to take a “functor” parameter:

- A functor is an object that simulates a function.
  - In this case, a functor to compare two objects would be expected by the sort routine.
  - The application code would define functor classes that compare students by name and by gpa.

Functors will be discussed at length in a few weeks.

41