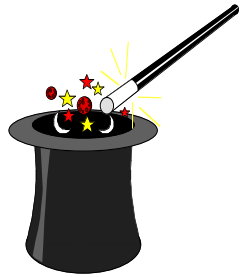
1

Java

• Was designed so that executables are portable. Java applications do not need to be "ported" to every software and hardware platform.

• Java "applets" are mini-applications which run inside a web page. Applets perform tasks or interact with the user on the browser page. Applets don't use resources from the web server once they are downloaded.

• Java is also a modern, general-purpose programming language which offers garbage collection, exception-handling and multi-threading within an object-oriented framework.



@1998 John Mylopoulos

2

Classes, Objects and Instantiation

• A class is a template, can have objects as instances all of which have the same structure (defined by the template)

• Classes are useful because they can be reused, possibly after extension, generalization, specialization or other forms of reshaping.

• A class is a cross between a type and a module

A module because it consists of a data part and an operations part

A type because it has an associated set of objects (its instances) in the same sense that a type has an associated set of values

Class is a run-time concept, type a compile-time concept

• Every object must be an instance of some class

• Not all variables are object-valued in Java (or other object-oriented languages, except in Smalltalk)

@1998 John Mylopoulos

3

An Example of a Simple Class

```
class Body {
 // talking about celestial bodies here
   public long IDNum;
   public String nameFor;
   public Body orbits;
           // around which it orbits

   public static long nextID = 0;
}
```

• All fields of this class are declared to be `public`, i.e., visible from outside. Their types are respectively `long` (number), `String`, `Body` and `long`.

• `Static` fields (or, class variables) are shared by all instances of a class. `nextID` keeps track of the next available ID that can be used for a `Body` object.

@1998 John Mylopoulos

4

Instantiating Classes

```
Body sun: new Body();
sun.IDNum = Body.nextID++;
sun.nameFor = "Sol";
sun.orbits = null;

Body earth: new Body();
earth.IDNum = Body.nextID++;
earth.nameFor = "Earth";
earth.orbits = sun;
```

This declares two variables which take `Body` objects as values, and initializes their respective fields. Note that `nextID` has to be incremented after its current value has been assigned to a `Body` object.

@1998 John Mylopoulos

5

Constructors

```
class Body {
   public long IDNum;
   public String nameFor;
   public Body orbits; // around which it orbits

   public static long nextID = 0;

  Body() {
     idNum = nextID++;
   }
}
```

Constructors have the same name as the class for which they create instances. Constructors are invoked after the instance variables of the new object have been initialized and after their explicit initializers have been executed.

@1998 John Mylopoulos

6

Constructors with Parameters

```
Body(String bodyName,Body orbitsAround) {
   this();
   name = bodyName;
   orbits = orbitsAround;
}
```

The parameterized constructor calls its homonymous cousin and then does some additional work. Now we can create two `Body` objects through the following:

```
Body sun = now Body("Sol", null);
Body earth = new Body("Earth", sun);
```


@1998 John Mylopoulos

7

Methods

Methods provide the code that understands and manipulates an object's state. Here is a method which constructs a string description for a `Body` object:

```
public String toString() {
   String desc = idNum + " (" + name + ")";
   if (orbits != null)
      desc += " orbits " + orbits.toString();
   return desc;
}
```

`toString` is a special method because it is called every time an object participates in a `String` expression.

```
System.out.println("Body " + sun);
System.out.println("Earth " + earth);
```

prints

```
Body 0 (Sol)
Earth 1 (Earth) orbits 0 (Sol)
```

Method parameters are passed by value, except for object values which are passed by reference.

@1998 John Mylopoulos

8

Using Methods to Control Access

```
class Body {
   private long idNum;
   public String nameFor;
   public Body orbits;
           // around which it orbits

   private static long nextID = 0;

  Body() {
     idNum = nextID++;
   }

   public long id()    {
     return idNum;
   }
}
```

This definition of `Body` hides the two fields `idNum` and `nextID` and offers a method named `id` for accessing `idNum`.

@1998 John Mylopoulos

9

The `main` Method

When you run a Java program, you must specify a class which drives the application. Given such a class, the system locates and runs the main method for that class. For example,

```
class Echo {
  public static void main(String[] args)
  {  for (int i=0; i<args.length; i++)
       System.out.print(args[i] + " ");
       System.out.println();   }}
```

Assuming this class is stored in ascii file `Echo.java`, it can be compiled with

```
javac Echo.java
 // bytecode file stored in Echo.class
```

and then invoked with

```
java Echo in here
```

where `java` is the bytecode interpreter, and `in here` are `args` to the `main` method. The output of this execution is

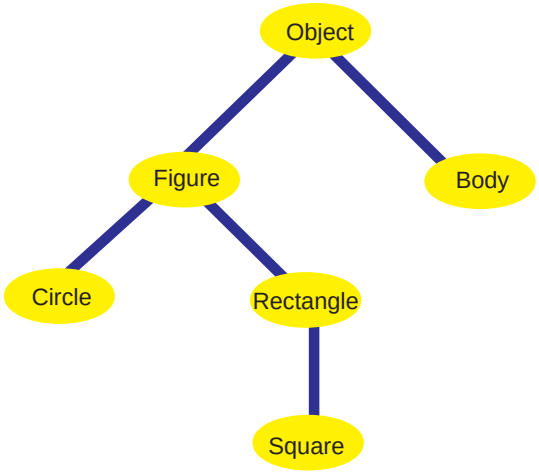
```
in here
```


@1998 John Mylopoulos

10

Extending Classes

A Java program can be thought as a collection of classes organized into a taxonomic hierarchy according to the generality/specificity of each class. `Object` is always the most general class in the hierarchy.



@1998 John Mylopoulos

11

The `Object` Class

The `Object` class is the root of the Java class hierarchy. This class declares methods that are implemented by all objects.

Variables of type `Object` can refer to any object, whether it is a class instance or an array.

The methods provided by `Object` have as follows:

```
public boolean equals(Object obj)
   // checks for equality
protected Object clone()
   // creates a clone object
public final Class getClass()
   // returns the class of an object;
   // classes are objects too, and are
   // instances of Class
protected void finalize()
   // reclaims an object
```


12

An Aside: Genericity

How can we define a `stack` class that will take elements other than strings?

Solution (Eiffel): Define a parameterized class `stack(type)` which is then instantiated

```
new stack (string), s1
s1 -> push ("hello")
new stack (int), s2
```

In Java, you declare `stack` to take elements of type `Object`. This way, the instances of `stack` would not be able to take primitive types as elements (`int`, `boolean` etc.), but you can make objects of these types by using "wrapping classes".

The Class `Attr`

Class `Attr` has attribute-value pairs as instances:

```
class Attr {
   private String name;
   private Object value = null;
   public Attr(String nameOf) {
      name = nameOf;    }
   public Attr(String nameOf,
               Object valueOf) {
      name = nameOf;
      value = valueOf     }
   public String nameOf()  {
      return name;    }
   public Object valueOf()  {
      return value;    }
   public Object valueOf(Object newValue)
   {  Object oldVal = value;
      value = newValue;
      return oldVal;
   }
}
```

An Extended Class

The class `ColourAttr` extends `Attr` for attributes whose value is a colour. Colours are represented as strings (e.g., "red", "green") or as decoded representations of type `ScreenColour` (defined elsewhere)

```
class ColourAttr extends Attr {
 private ScreenColour myColour;
                 // the decoded colour
 public ColourAttr(String name) {
      this(name,"transparent");  }
 public ColourAttr(String name,
                       Object value) {
  super(name,value);
  decodeColour();     }
 public ColourAttr(String name,
                   ScreenColour value) {
  super(name,value.toString()));
  myColour = value;     }
```

15

More...

```
public Object valueOf(Object newValue){
  Object retval = super.value(newValue);
  decodeColour();
  return retval;     }
 public ScreenColour valueOf(
             ScreenColour newValue)  {
  super.valueOf(newValue.toString());
  myColour = newValue;
  return newValue;     }
 public ScreenColour colour() {
  return myColour;  }
 protected void decodeColour() {
  if (valueOf() == null) myColour = null;
  else myColour =
        new ScreenColour(valueOf());
   }
}
```

This class overloads and overrides the `valueOf` methods inherited. It also provides a new `colour` method to return the colour description of the `colour` field.

@1998 John Mylopoulos

16

Overriding and Overloading Methods

Overloading a method means that we have more than one method with the same name, but with different signatures. For example,

```
public ColourAttr(String name)
public ColourAttr(String name,
                  Object value)
public ColourAttr(String name,
                  ScreenColour value)
```

Overriding a method means replacing the superclass' implementation with one of your own. The signatures of the two methods must be identical. Moreover, only non-static methods can be overridden.

@1998 John Mylopoulos

17

Polymorphism and Overloading

• Polymorphism means having many forms; in programming, polymorphism arises in a variety of situations:
  - a variable may be allowed to hold values of different types
  - a function may take arguments of different types, e.g., area for different types of geometric figures
  - a function may take different number and types of arguments, e.g., minus as a unary function that returns the negative of its argument or minus as a binary arithmetic operation
  - a data type may be parameterized; say, list of T, where T is a parameter

• Overloading means that the same name is used for several different things, for example different functions having the same name, or different objects having the same name.

Overloading and polymorphism are similar concepts unconstrained overloading is a (probably least desirable) special case of polymorphism

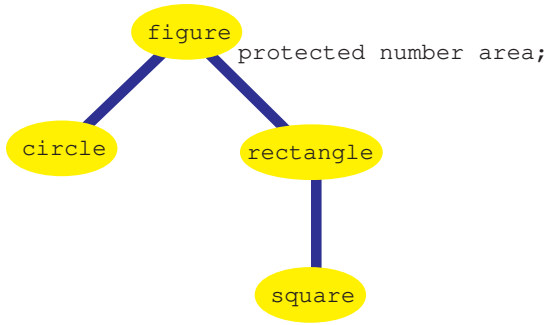
@1998 John Mylopoulos

18

`Protected` Class Members

If a class member (method or field) is protected, then it can only be used by classes which extend that class.

For example,



Here `area` is protected. This means that the code in `rectangle` can access `area` only through a reference to a type that is a `rectangle` or a `square` but not a `circle`.

`Protected` is a compromise between `public` and `private`.

@1998 John Mylopoulos

19

Hiding Fields

Fields can't be overriden, but they can be hidden by re-declaring a field defined in a class in one of its sub-classes. For example,

```
class SuperShow {
   public String str = "SuperStr";
   public void show() {
    System.out.println("Sup.show: " + str);
      }  }

class ExtendShow extends SuperShow {
   public String str = "ExtendStr";
   public void show() {
    System.out.println("Ext.show:" + str);
      }  }

public static void main(String[] args){
 ExtendShow ext = new ExtendShow();
 SuperShow sup = ext;
 sup.show();
 ext.show();
 System.out.println("s.str="+sup.str);
 System.out.println("e.str="+ext.str);
}
```


20

Hiding Fields (cont'd)

The output of this program will be

```
Ext.show: ExtendStr
Ext.show: ExtendStr
s.str= SuperStr
e.str= ExtendStr
```

The first two outputs suggest that whenever `show()` is called, it is the most specialized `show` method that is actually executed.

The last two outputs suggest that the object that has been created has two `str` fields, one for `SuperShow` and one for `ExtendShow`.

21

The `super` Keyword

`super` accesses the methods of an object's superclass. `super` can only be used with non-static methods of an extended class.

Note that selection of a method through `super` acts differently than selection through an object's class:

```
class That {
 protected String nm() { return "That"; }}
class More extends That {
 protected String nm() { return "More"; }
 protected void printNM() {
  That sref = super;
     // sref refers to the same object as
     // an instance of That
  System.out.println("this.nm() = "
                          + this.nm());
  System.out.println("sref.nm() = "
                          + sref.nm());
  System.out.println("super.nm() = "
                          + super.nm());
```

What does `printNM` output?

```
this.nm() = More
sref.nm() = More
super.nm() = That
```


@1998 John Mylopoulos

22

Marking Methods and Classes `final`

Marking a method `final` means that no extended class can override it to change its behaviour. Entire classes can be marked `final` too:

```
final class NoExtensionsPlease {
   ...        }
```

in which case it can't be subclassed.

Why use `final`? Firstly, it enhances security. Anyone who uses a `final` class knows that its behaviour can't be changed. In addition, `final` simplifies optimizations done by the Java run-time environment.

@1998 John Mylopoulos

Abstract Classes and Methods

Declaring an abstract class means that you only need to define part of the implementation, while the rest is to be provided by subclasses.

For example, you could define

```
public abstract class figure {
   public abstract Number area(); }

public class circle extends figure {
   public Number radius:
   public Number area() {
      return pi * radius * radius; }
```

Abstract classes can't be instantiated directly, only their subclasses can.

Sometimes it is useful to declare a subclass or one of its methods `abstract`, thereby "turning off" implementations inherited from higher up.

Interfaces

An interface declares a type consisting of abstract methods and constants (`static`, `final` fields), independently of any implementation. For example, suppose you wanted to use the `Attr` class in various other classes. First, you define an interface `Attributed`

```
interface Attributed {
   void add(Attr newAttr);
   Attr find(String name);
   Attr remove(String name);
   java.util.Enumeration attrs();  }
```

This interface declares four abstract methods for adding/removing an attribute, checking if an attribute exists, and listing all attributes of an object.

We can now extend the `Body` class to allow celestial objects to have associated attributes:

```
class AttributedBody extends Body
   implements Attributed
```

This way, `AttributedBody` inherits the methods of the interface, even though only one class is extended (`Body`).

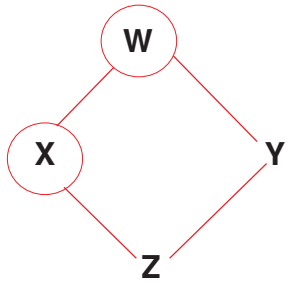
25

Single vs Multiple Inheritance

Java allows each class to have a single superclass, but possibly multiple interfaces. For example,

```
interface W {}
interface X extends W {}
class Y implements W {}
class Z extends Y implements X {}
```

Here we have multiple inheritance for interfaces, but single implementation inheritance



@1998 John Mylopoulos

26

Multiple Inheritance and Name Conflicts

The inheritance hierarchy for interfaces, unlike classes, does not have a single root.

You can have classes which only extend interfaces:

```
interface W {}
interface X extends W {}
interface Y extends W {}
class Z implements X, Y  {}
```

If there are name conflicts for methods, for example interfaces `X` and `Y` above use a method named `myMethod`, then there may a problem disambiguating in `Z`. If the synonymous methods have different signature, i.e., different parameters and/or returned values, it is possible to disambiguate. Otherwise the methods will be treated as identical and will afford a single implementation.

If constant `const` appears in both `X` and `Y`, then we have to refer to each through `X.const`, `Y.const` within `Z`, for disambiguation purposes.

@1998 John Mylopoulos

27

Implementing Interfaces

You may have several implementations for the `Attributed` interface (depending on how many attributes you expect to deal with). Here is one:

```
import java.util.*

class AttributedImpl
        implements Attributed {
 protected Hashtable attrTable =
                   new Hashtable();
 public void add(Attr newAttr) {
  attrTable.put(newAttr.nameOf(),
                         newAttr); }
 public Attr find(String name) {
   return(Attr)attrTable.get(name); }
 public Attr remove(String name) {
  return (Attr)attrTable.remove(name);}
 public Enumeration attrs() {
   return attrTable.elements(); }}
```


@1998 John Mylopoulos

28

Using an Implementation

You can use an implementation by extending a class. But if you want to use several implementations, you need a different approach. One way to do it is to create an object of an implementation type and make it part of the state of the class which uses the implementation:

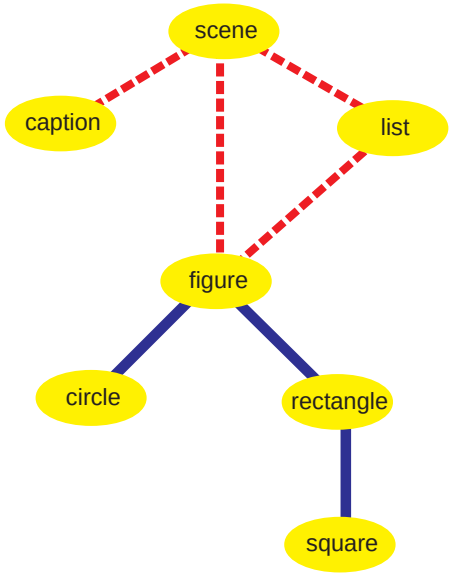
```
import java.util.Enumeration;

class AttributedBody extends Body
      implements Attributed {
 AttributedImpl attrImpl =
                new AttributedImpl();
 AttributedBody() { super();}
 AttributedBody(String name,Body orbits){
      super(name,orbits); }
   // forward all Attributed calls to the
   // attrImpl object
 public void add(Attr newAttr)
   {attrImpl.add(newAttr);}
 public Attr find(String name)
   {return attrImpl.find(newAttr);}
 public Attr remove(String name)
   {return attrImpl.remove(name);}
 public Enumeration attrs()
   {return attrImpl.attrs();} }
```


@1998 John Mylopoulos

29

Part-of and Is-a



A scene consists of (has parts) a list of figures and a caption. Each of the figures in the list is itself a part of the scene.

@1998 John Mylopoulos

30

Interfaces and Abstract Classes

There are two important differences between them:

• Interfaces support multiple inheritance, whereas classes only support single inheritance;

• An abstract class can have a partial implementation, but an interface can't have any implementation information.

You want to use interfaces when multiple inheritance is important. You want to use abstract classes if you want to fix at least partly the implementation to be inherited by several other classes.

@1998 John Mylopoulos

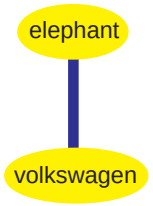
Different Kinds of Subclasses

• Specialization -- the children are specialized versions of the parent class, e.g., `square is-a rectangle`. Here the parent class is fully defined, its definition is possibly refined for the subclasses

• Extension -- the children extend the behaviour of the parent class in some way, as with `extendedStack`

• Arbitrary modification -- the definition of the parent class is overriden to define an altogether different class

don't use subclassing for this reason!



• Specification -- here the parent class simply defines the syntax and structure of subclasses, leaves the rest to subclass definitions, e.g., `figure`

• Combination (mixin') -- for languages that allow multiple parent classes to a class, mixin allows the combination of structure and behaviour of two parent classes

@1998 John Mylopoulos

Cloning Objects

The `Object.clone` method helps you write clone methods for your own classes. A clone method returns an object whose initial state is a copy of the current state of the object on which the `clone` method was applied.

There are three major considerations in writing a `clone` method for a particular class:

• Implementation of the `cloneable` interface:

```
interface cloneable {
 Object clone()
  throws CloneNotSupportedException;}
```

• the `Object.clone` method which copies all the fields of the original object to the new object; this works for many classes, but may need to be supplemented for particular classes;

• the `CloneNotSupportedException` exception, which is used to signal that a class' `clone` method should not have been used.

@1998 John Mylopoulos

33

Implementing the `Cloneable` Interface

If you are cloning objects whose fields have scalar values, there is no problem. However, if you are cloning objects whose fields have `object` or `array` values, then things can get messy because the `Object.clone` method does not copy `object` or `array` values, but only the references to them.

```
public class IntStack
           implements Cloneable {
 private int[] buffer;
 private int top;
 public IntStack(int maxContents) {
  buffer = new int[maxContents];
  top = -1;  }
 public void push(int val) {
  buffer[++top] = val;  }
 public int pop() {
  return buffer[top--]; }
}
```

This implementation of `Cloneable` uses the `Object.clone` method inherited from `Object`.

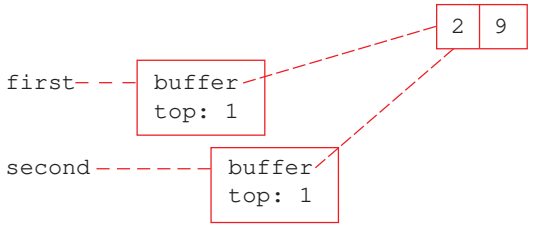
34

The Problem with Cloning

Suppose now you create an `IntStack` object and you clone it:

```
IntStack first = new IntStack(2);
first.push(2);
first.push(9);
IntStack second = (IntStack)first.clone();
```

What have you got?



Doing Cloning Right

The right way of doing cloning here is to override the `Object.clone` method:

```
public Object clone() {
 try {
 IntStack nObj = (IntStack)super.clone();
  nObj.buffer = (int[])buffer.clone();
  return nObj;  }
 catch (CloneNotSupportedException e) {
  // can't happen here; we support
  // clone, and so do arrays
  throw new InternalError(e.toString());
                                } }
```

Now you first use the `clone` method of the superclass to copy all fields, then you assign a new value to the `buffer` field by cloning the array value.

For things to work properly, some type conversion is required. For example, `super.clone` may return an `Object`, and therefore we want to change its type to `IntStack`.

@1998 John Mylopoulos

Classes and Cloning

A class can have one of four attitudes towards cloning:

- Supports cloning by implementing the `cloneable` interface, as discussed;

- Supports cloning conditionally. Such a class may be a collection class, whose instances are collections of elements; its instances can then be cloned in principle, but only if the elements of a collection can be themselves cloned;

- Allows subclasses to support `clone`, but doesn't support it itself; this is done by not implementing `cloneable`, but by providing a `clone` method which clones its own fields (to be used by subclasses);

- Forbids cloning by not implementing `cloneable` and by defining `clone` so that it always throws up the exception`CloneNotSupportedException`.

@1998 John Mylopoulos

Exceptions

Most programs don't check for all possible error conditions that could be encountered, such as division by `0`, because this is time-consuming and tends to clutter one's program.

Exceptions provide a clean mechanism for dealing with error conditions without facing the dilemma of cluttering one's program or expect frequest crashes.

Exceptions make the error conditions that can be signalled by a method part of the method's contract. These exceptions, can be seen by the programmer, checked by the compiler and respected by extended classes which override the method.

An exception is thrown when an unexpected error occurs. The exception is then caught by an encompassing clause up the method invocation stack. If an exception is not caught, a default exception handler takes effect, usually printing useful information about where the exception was thrown.

@1998 John Mylopoulos

Creating Exception Types

Exceptions are objects and are instances of the class `Exception` or `Throwable`, or one of their subclasses.

Exceptions can have associated information which describes the circustances where the exception was raised. For example, suppose we add a `replaceValue` method to the `Attributed` interface. If this method is called for an object that doesn't have the attribute in question, an exception is raised:

```
public class NoSuchAttrException
                extends Exception {
 public String attrName;
 public Object newValue;
 NoSuchAttrException(String name,
                     Object value) {
  super("No attr named \"" +
              name + "\" is found");
  attrName = name;
  newValue = value;   }
}
```

This exception subclass includes a constructor which assigns the two fields of every `NoSuchAttrException`.

@1998 John Mylopoulos

`throw` and `try`

Exceptions are thrown by using the `throw` statement:

```
public void replaceValue(String name,
                         Object newValue) {
   throws NoSuchAttrException
 Attr attr = find(name);
 if (attr == null)
   throw NoSuchAttrException(name,this);
 attr.valueOf(newValue);
}
```

The exception thrown by `replaceValue` can be caught by a method which includes the `try` clause:

```
try {
 attributedObj.replaceValue("Age",
                    new Integer(8)); }
   catch (NoSuchAttrException e) {
  // shouldn't happen, but recover anyway
   Attr attr =
      new Attr(e.attrName,e.newValue);
   attributedObj.add(attr); }
```

Notice that a different caller of `replaceValue` may have a different exception handler which prints a message and aborts.

@1998 John Mylopoulos

General Form of `try`

```
try
   block
catch (exceptionType identifier)
   block
catch (exceptionType identifier)
   block
...
finally
   block
```

The body of `try` is executed until an exception is caught which matches one of the declared exception types. When this happens, the corresponding `catch` block is executed.

The `finally` block is executed after all other processing inside the `try` method. Then control returns to the statement after the `try` statement.

@1998 John Mylopoulos

41

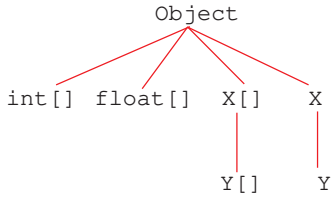
Arrays

Java arrays provide ordered collections of elements (a lot like lists do in Lisp and Prolog).

```
int [] ia = new int[3]
```

creates an integer array with three elements. An array's length is fixed at creation time. Indices run from `0` to `length(array) - 1`.

Arrays are implicit extensions of `Object`. If `Y` is a subclass of `X`, then `Y[]` is a subclass of `X[]`:



42

Arrays of Objects and Arrays

When you are declaring an array of objects, you are actually declaring an array of variables:

```
Attr[] attrs = new Attr[12];

for (int i = 0; i<attrs.length; i++)
   attrs[i] = new Attr(names[i], values[i]);
```

Here, after initialization you have an array of 12 variables initialized to `null`.

You can have arrays of arrays:

```
float[][] mat = new float[4][4];
setupMatrix(mat);
for (int y=0;y<mat.length;y++)
 for (int x=0;x<mat[y].length;x++)
  System.out.print(mat[x][y]+" ");
  System.out.println();
```

This declares, sets up and then prints a 2-dim array. The first dimension of an array must always be defined, others may be left unspecified.

Packages

Packages group together related classes, interfaces and sub-packages.

For example, we may want to make a package of our attribute classes. To do so, we declate at the beginning of the source code file

```
package attr
```

in which case all class and interface declarations in this file belong to the `attr` package.

To use a declaration from within this package somewhere outside, you can prefix every usage with `"attr."`. Alternatively, you can `import attr.*`, in which case all declarations associated with the package are imported in another package.

Classes and interfaces within a package may be declared to be `public` or `package`.

Every file of source code is treated as a package, possibly unnamed.

@1998 John Mylopoulos

Wrapper Classes

Wrappers are used to define objects whose internal state includes a data structure, so the objects "wrap" the data structure.

For example, we have seen that in order to define generic types, such as `Stack`, we declared the contents of a `Stack` object to be `Object`. If we want a stack of integers, we need a wrapper class for integers.

There are built-in wrapper classes for all primitive types, including `int`, `char`, `String` etc.

Every wrapper class provides the following methods:

- A constructor which takes a primitive type and creates a wrapper object, e.g., `Number(4)`;
- A constructor which takes a string argument and creates an appropriate object, e.g., `Number("4")`;
- A `toString` method;
- An `intValue`, `charValue`, etc. method which produces the primitive type;
- An `equals` method;
- A `hashCode` method to return a hash code for use in hash tables.

@1998 John Mylopoulos

The Java Infrastructure

Java is designed to maximize portability. This means that platform-dependent language features, such as pointers, are left out, while others, such as numbers, are specified completely in the language.

Java source code is compiled into Java bytecodes, which is designed to run on a Java virtual machine. Bytecodes define a machine language for an abstract machine.

The virtual machine assigns each application its own runtime. This makes it possible to isolate applications from each other and from the host operating system. The security manager may, for example, forbid the application from reading from the local disk.

@1998 John Mylopoulos