Java: origin

- In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices
- They wanted the language to be hardware independent since different manufacturers would use different CPUs.
- These conditions led them to decide to compile the code to an intermediate machine-like code for an imaginary CPU called a virtual machine. This intermediate code (called bytecode) is completely hardware independent.
- Programs are run by an interpreter that converts the bytecode to the appropriate native machine code. Thus, once the interpreter has been ported to a computer, it can run any bytecoded program.
- This project had a lot of trouble getting others interested in Java for smart devices.
- It was not until they decided to shift gears and market Java as a language for web applications that interest in Java took off.

Java is Object-Oriented. A key concept is that of **class**

A class definition in Java looks like:

```
class NAME
{
   --- fields ---
   --- constructors ---
   --- methods ---
}
```

Roughly: The fields are the data structures that define the data type. Constructors are used to initialize newly created instances. Methods are the users' gateway into the class.

Let’s look at an example:

```
class Pos
{
  // fields:
  float lati, longi;

  // constructor:
  Pos(float l1, float l2)
  { lati = l1;
    longi = l2;  }

  // methods:
  float get_lati()
  {    return lati;  }

  float get_longi()
  {    return longi;  }

  Pos add(float a, float o)
  {    Pos new_pos = new Pos(lati + a, longi + o);
    return new_pos;    }
}
```

This defines a class of objects representing a position on the surface of the earth. A position is defined by a latitude and longitude, each a floating-point (i.e., noninteger) number. To make a new pos, a program using this class might execute:

```
Pos p1;
...
p1 = new Pos(30, 170);
```

There are occasions when one needs to refer to an object from within one of its methods. The variable `this` is always bound to the object that "owns" the method it occurs in

Note that `p1.lati` and `p1.get_lati()` return the same value.

Better to flag `lati` as `private`, so that it is not visible outside the class.

To create data constants, use keywords `static final`

One class can extend another, meaning that it **inherits** the functionality of that class, modified in some way. For instance, suppose you want a kind of Pos whose latitude is always zero:

```
class Equatorial_Pos extends Pos
{
  Equatorial_Pos(float g)
    {
      lati = 0;
      longi = g;
    }

  float longi_diff(Equatorial_Pos other)
    {
      return longi - other.longi;
    }
}
```

Now you can create an equatorial `Pos` by writing, e.g.,

```
e = new Equatorial_Pos(130);
```

and access its longitude by writing

```
e.get_longi()
```

You don’t have to define these members again; they’re all inherited from the original `Pos` class.

`Pos` is said to be a superclass of `Equatorial_Pos`, which is a subclass of `Pos`.

The new method `longi_diff` takes the "difference" in some sense between two equatorial Pos’es. You would call it by writing

```
e1.longi_diff(e2)
```

This method cannot be used with variables bound to `Pos`’es, but only with elements of the subclass `Equatorial_Pos`.

There are three kinds of data type in Java: primitive types, object types, and array types.

Main the primitive types are `int` ("integer"), `float` (noninteger number), `boolean`, and `char` ("character").

Object types are instances of classes.

Arrays are indexed collections of entities. To create a two-dimensional array of integers:

```
...
int[][] a;
...
```

The dimensions are specified with an array is created, using new:

```
...
int i,j;
int[][] a;
...
a = new int[2][i+j];
...
```

This code makes a an array of 2 by i+j integers.

Arrays have subscripts starting at 0, which means that in my example the first subscript can be 0 or 1, and the second can be 0, 1, 2, ..., up to i+j-1.

We can also introduce the array thus:

```
int i,j;
...
int[][] a = new int[2][i+j];
```

To handle strings, Java supplies a built-in class called `String`.

To concatenate strings, you can write `string1 + string2`.

To get the nth character of a string s, write `s.charAt(n).`

Operators

for arithmetic:

```
&  + - * /
```

autoincrement/autodecrement:

```
++ --
```

shortcuts:

```
+= -=
```

equality/inequality:

```
== !=
```

and or:

```
&& ||
```

Control statements

```
if, while, for
```

When an array is created, how it gets initialized depends on whether its elements are primitives or objects. Primitives are initialized to 0, 0.0, false, or some other "nullish" value. Objects are initialized to an important constant called `null`.

initialization of an array. You have to loop:

```
...
Pos[][] p = new Pos[3][4];
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 4; j++)
     p[i][j] = new Pos(0,0);
```

Static Members: There are occasions when it makes more sense to attach a procedure to a class instead of to each element of a class.

Example of **static method**. Keyword: `static`; called by writing `classname.methodname(...)` instead of `object.methodname(...)`.

Exceptions

An abnormal circumstance is called an exception. It is, of course, just another kind of object.

The occurrence of an abnormal situation results in throwing an exception.

Java system finds some calling procedure that anticipated such a situation, and asks that procedure to handle it. This is called catching the exception.

```
\\ Here's an example:
   ...
   int i;
   FileReader fr;
   try
     {
       for (i = 0; i < 10; i++)
         {
           fr = new FileReader("Chapter " + i + ".doc");
           char c = fr.read();
           ...
         }
      }
   catch(FileNotFoundException e)
     {
       System.out.println("File " + i " not found; aborting");
       System.exit(0);
     }
   catch(IOException e)
     {
       System.out.println("Couldn't read from file " + i "; aborting");
       System.exit(0);
     }
   ...
```

The construct

```
try
  {
    Normal stuff;
    Hopefully nothing odd will happen here:
  }
catch(exception-class-1 e)
  {
    What to do if exception of class-1 occurs;
  }
catch(exception-class-2 e)
  {
    What to do if exception of class-2 occurs;
  }
```

This is a nice facility, but there is one annoying detail: If the compiler finds that one of your procedures can create an exception, then you must either catch it in that procedure, or amend the header of the procedure with the phrase "throws exception-class-1, exception-class-2, etc.."

One of the exempted exceptions is dividing by zero.

Interfaces

An interface is a named collection of method definitions (without implementations). An interface can also declare constants.

A class can implement many interfaces but can have only one superclass.

Use: specify a behaviour that a set of possibly unrelated classes should have.

Defining an interface is similar to creating a new class. An interface definition has two components: the interface declaration and the interface body.

```
interfaceDeclaration {
    interfaceBody
}
```

The interfaceDeclaration declares various attributes about the interface such as its name and whether it extends another interface. The interfaceBody contains the constant and method declarations within the interface.

A class that implements an interface adheres to the protocol defined by that interface.

Example of use:

```
public class St extends Ap implements SW {...}
```

(extending the abstract class `Ap` and implementing the interface `SW`)

Example of interface:

```
interface Lookup {
   /** Return the value associated with the name, or
    * null if there is no such value */
    Object find(String name);
}
```

Code that uses references of type `Lookup` (references to objects that implement the interface) can invoke the `find` method

```
void processValues(String[] names, Lookup table) {
   for (int i = 0; i ! names.length; i++) {
       Object value = table.find(names[i]);
       if (value != null)
          processValue(names[i], value);
   }
}
```

Here is a class that implements the previous interface using an array

```
class SimpleLookup implements Lookup {
   private String[] Names;
   private Object[] Values;

   public Object find(String name) {
      for (int i = 0; i < Names.length; i++) {
          if (Names[i].equals(name))
             return Values[i];
      }
      return null; // not found
   }
   // ...
}
```

Abstract Classes

An abstract class may contain both concrete and abstract methods, that is, methods with no implementation.

Example:

the graphic objects are also substantially different in many ways: drawing a circle is quite different from drawing a rectangle.

```
abstract class GraphicObject {
            int x, y;
            . . .
            void moveTo(int newX, int newY) {
                . . .
            }
            abstract void draw();
        }
```

GraphicObject also declares abstract methods for methods, such as draw, that need to be implemented by all subclasses, but are implemented in entirely different ways (no default implementation in the superclass makes sense)

GraphicObject however provides member variables and methods that are wholly shared by all subclasses, such as the moveTo method

Examples of use:

```
class Circle extends GraphicObject {
            void draw() {
                . . .
            }
        }
class Rectangle extends GraphicObject {
            void draw() {
                . . .
            }
        }
```

Program Structure

Java supports some very complex but elegant control structures, including multithreading, programs on web pages ("applets"), and communication over the Internet.

But for now we ignore all the hairy stuff.

A Java program consists of one or more text files with extension ".java" . You compile a program called yourprog.java by executing

```
javac yourprog.java
```

This will generate several class files, which have extension ".class"

There must be a class called yourprog, with a "main procedure":

```
class yourprog
{
  public static void main(String[] args)
  {
    ...
  }
}
```

This is the procedure that is called when you run your program, that is, when you execute

```
java yourprog
```

in a command shell.

Another consequence of main being static is that any variable it refers to must either be local to the main procedure itself, or a static variable of its class.

The main method must be defined as

```
public static void main(String[] args)
```

In the Java application programming interface, classes are grouped together into **packages**.

To use all classes in the package util, do

```
import java.util.*
```