JAVA

Developed in the beginning of the 1990's by James Gosling from Sun Microsystems. Formally announced in a major conference in 1995.

Java programs are translated (compiled) into **Byte Code** format.

- The byte code is executed by a program that pretends its a computer based on the byte code instruction set (**Byte Code Interpreter**).

- A byte code interpreter intended to execute the byte code produced by the Java Compiler is called a **Java Virtual Machine**.

Java vs. C++

1. Java programs are **compiled** into **bytecodes** which are executed by the Java Virtual machine (**Interpreter**)

2. Used to build stand alone applications or application that are executed by an **applet** in a Web page

3. **Garbage Collection**

4. Method declaration and implementation are together

5. No Multiple Inheritance

6. Use of **Interfaces**

7. No overloading of operators

8. No pointers (use **new** for new objects)

9. Everything is an object of a class except the primitive data types

10. Class Wrappers for primitive data types

11. Primitive data types are passed by value and objects by reference.

12. Real **Arrays**, class **Vector**

13. Class **Boolean**

14. All classes are subclasses of **Object**

15. Pseudo variable **super**

16. Access modifiers (***public, protected, private***) are applied individually to every declaration

17. Multithreading or concurrency

18. Keyword **final**, **abstract**

Java programs can be created two ways:

1. Applications: stand alone programs that can be invoked from the command line.

2. Applets: program embedded in a web page (not intended to run on its own), to be run when the page is browsed.

Application Example:

```
import java.io.*;

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

- The source code is saved in a file called **Hello.java**
- To compile: **javac Hello.java.** After compilation a new file called **Hello.class** is stored in the directory.
- To run: **java Hello**

Applet Example:

```
import java.applet.*;
import java.awt.*;

public class Hello extends Applet  {

    public void paint(Graphics g)  {
       g.drawString("Hi", 20, 10);
    }
}
```

To use this applet in a Web page, we have two tasks to perform:

1. Compile the Java source code into byte code (same as before)

2. Prepare an HTML document which describes the Web page in which the applet will live.

- Create the **myPage.html** document

```
<html>
<applet code = "Hello.class"
                    width = 25  height = 35>
</applet>
</html>
```

- Start the Web browser and open myPage.html

Primitive Data Types

- Integers (byte, short, int, long)

- Floating Point (float, double)

- Characters: Unicode (16 bit unsigned)

- Boolean (false, true)

Ex:
```
boolean b;
if (b == true) . . .
if (b) . . .
System.out.println(b)
```

- All variables have a standard default value

Strings

- The type String is a class (not primitive) but it is important enough to have support for a couple of features built into the language.

- A string literal is zero or more chars enclose in double quotes

  ```
  " "          "Example"
  ```

- Although String is a class you can create a new literal instance:

  ```
  String p = "An Example";
  String p = new String("An Example");
  String p = "Example" + "of a long" + "literal";
  ```

- The + operator will concatenate any primitive type to a String. If you supply any object as the operand the system will call a method **toString()** to create a string representation of that object and then concatenate.

  ```
  int n = 10;
  String s = "Result" + n + "is OK";
  ```

- For any class you can provide a method public **String toString()** or you can inherit the method from class **Object**. This will return a string containing the class name and something that looks like an address of the instance.

  ```
  Bird eagle = new Bird();
  String s = "The Bird is an" + eagle;
  ```

- Java strings are immutable: once you create it you can't change it. However you can easily cause a string variable to refer to another string.

Arrays

- Like in C++ array subscripts are from **zero** to **(length - 1).**
- You can not subclass or extend an array.
- You can not define your own methods for arrays.
- To determine the number of elements of an array use the instance variable **length**. For strings you use the method **length().**
- You can declare and initialize arrays of primitive and reference types:

Example:

```
int scores[] = new int[4];
int values[] = {65, 87, 60, 93, 45};


class Student {
  public int x1, x2;
  Student(int n1, int n2) {
    . . .
  }
  . . .
}


Students scores[] = new Student[4];
Student Z[] = {  new Student(90,72), new Student(86, 98), . . . }
```

- Declarations do not create objects

```
Bird b[] = new Bird[4];

Bird eagle = new Bird();
b[2] = eagle;
b[3] = new Bird();
```

- Arrays with the same element type, and the same number of dimensions, can be assigned to each other. The arrays do not need to have the same number of elements because what actually happens is that one reference variable is copied into the other.

```
int eggs[] = {1, 2, 3, 4};
int ham[] = {1};
ham = eggs;
ham[3] = 0;     // now ham has 4 elements
```

Example: Array Copy

```
char copyFrom = {'d', 'e', 'c', 'a', 'f',
                 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd'};
char copyTo = new char[7];
System.arrayCopy(copyFrom, 2, copyTo, 0, 7);
```

- The size of an array is an expression that can be evaluated at run time. However, once allocated it cannot be resized!

```
int [] numbers = new int[n];
```

- The [ ] is checked at run time. When and invalid index is found, an **exception** is thrown that terminates the program unless explicitly handled in some other way.

Compilation Units and Packages

Java does away with all the aggravation of header files, forward declarations, and ordering declarations. You must thing of javac as a two-pass compiler:

1. On the first pass, the source code is scanned, and a table built of all the symbols.

2. On the second pass, the statements are translated and since the symbol table already exists, there is never any problem about needing to see the declaration of something before its use. It's already in the symbol table.

Thus, any field in a class can reference any other field, even if it does not occur until later down the page or in another file.

There are two simplifications in Java programs regarding class definitions and files (less flexibility, but more simplicity):

1. There are no nested class definitions.

2. Everything to do with a single class goes in one file. You can put several related classes in one source file, but you cannot go the other way. This ensures that when you look in a source file, you get the class, the whole class, and nothing but the class (and its buddies).

A source file is what you present to a Java compiler, so the contents of a complete source file are also known as a Compilation Unit.

Automatic Compilation

Java tries hard not to let you build a system using out-of-date components. When you invoke the compiler it looks to see what other classes it references. If the class that you are compiling makes references to a class in another file if can notice if:

1. The second .class file does not exist but the **.java** does.

2. The **.java** file has been modified without recompilation.

In both of the above cases, it will compile the other **.java** file.

There is a related **"-cs"** option when running an application to check that the source has not been updated after the bytecodes were generated. If the source is newer, it will recompile it.

java -cs myApplication

File Name is Related to Class Name

Java emphasizes the idea that the name of a file is related to its contents. A compilation unit (source file) has several rules:

1. It can start with a "package" statement, identifying the package (a library) that the byte codes will belong to.

```
package myDir;
```

2. Next can come zero or more "import" statements, each identifying a package or a class from a package, that will be available in this compilation unit.

```
import java.io.*;
```

3. The rest consists of Class declarations and Interface declarations.

4. At most one of the classes in the file can be "public". This class must have the corresponding name as the file it is in.

```
public class plum      //must be in file plum.java
```

Each of the classes in a **.java** file will create an individual **.class** file to hold the generated byte codes. If you have three classes in a source file, the compiler will create three **.class** files.

Packages

Java allows to group files into packages. A package is a collection of individual .class files in a directory (both a directory and a library). Makes classes easier to find and use, and helps avoid name conflicts. To indicate that a file is part of a package, all you need is to include as the very first line:

```
package packageName;
```

and then make sure that the file is in the directory or folder with the same name as the package. Package names must match the directory name.

Example: Suppose we created two widget classes: GraphicsButton and Slider. For convenience we might have put them in separate files, but we want to include them in a package.

```
//----- File GraphicsButton.java ---


package myWidgets;

import java.awt.*;              // to import Canvas

public class GButton extends Canvas
        {. . .}
```

```
//----- File Slider.java --------------


package myWidgets;

import java.awt.*;

public class Slider extends Canvas
        {. . .}
```

```
//------ FileTest.java ----------------


import java.applet.*;

import java.awt.*;

import myWidgets. *;

public class Test extends Applet
  {
     GraphicsButton myButton;

    Slider mySlider;

  . . . }
```

Import allows you to use a class name directly, instead of fully qualifying it with the package name.

```
Class Pie {
        java.util.Date D1;
        double w;
}
```

```
import java.util.Date;

Class Pie {
        Date D1;
        double w;
}
```

The package name can also be used to avoid name conflicts. If by some chance the name of a class in one package is the same as the name of a class in another package, you must disambiguate the names by prepending the package name to the beginning of the class.

Example: If we defined a **Rectangle** class in the **myWidget** package. The **java.awt** package also contains a Rectangle class. If both packages are imported then you have to be more specific and indicate which Rectangle class you want.

```
myWidgets.Rectangle rect;
```

Package names can have multiple components separated by periods representing directories in the file system.

```
package java.code.examples
```

Access Modifiers

Private: Can an object of type Alpha access private members of another Alpha object ?

```
class Alpha  {
        private int n;
        boolean isEqualTo (Alpha anAlp)
          {
              if (n == anAlp.n)
                    return true;
              else
                    return false;
          }
}
```

Perfectly legal. Objects of the same type have access to one another's private members. This is because access restrictions apply at the class or type level rather than at the object level !

Protected

```
package Greek;

class Alpha {
        protected int p;
        protected void proMet()  {
          System.out.println("protected");
        }
}
```

```
package Greek;

class Gamma {
        void accessMet()  {
                Alpha a = new Alpha();
                a.p = 10;
                a.proMet();
        }
```

```
package Latin;

import Greek.*;

class Delta extends Alpha {

	void accessMet(Alpha a, Delta d)
	{
		a.p = 10;
		d.p = 10;
		a.proMet();
		d.proMet();
	}
}
```

The Delta class can access protected members but only on objects of type Delta or its subclasses.

- If a class is both a subclass of and in the same package as the class with the protected member, then the class has access to the protected members.

- You can combine the virtues of private and protected by making an instance variable private and having protected get() or set() methods.

Static Members

- The **static** keyword can be used with data members and methods.
- Static variables and methods are called **class variable**s and **class methods** because they are associated with the class and not with any instance of the class.
- Inside the class, static members are accessed by giving its name.
- Outside the class the can be accessed using the name of the class or the name of an object.
- Many prefer to use the class name so it is clear that they are referring to a static member.

Example:

```
class Employee {
        String [] name;
        float salary;
        static int totalEmployees;
        . . .
        static void clear() {
                totalEmployees = 0;
        }
}

Employee rookie = new Employee();
rookie.totalEmployees ++;
Employee.totalEmployees  = 2;
rookie.clear();
Employee.clear();
```

- Static methods cannot access any instance data.

- Public class variables defined in a public class are accessible from any point in a program.

- Class methods are commonly used in classes like **Math** that are used only to hold methods, and are not subclassed. Such classes often have no constructor and all their methods are static.

```
Y = Math.sqrt(x);
```

- Class variables are often used with **final** to define constants. This is more efficient than **final** instance variables because constants can't be changed so you really need one copy.

Final keyword

- If you want a **variable** to be a constant, add the keyword final, and initialize the variable in the declaration (uppercase by convention)

- If you mark a **class** with final, then it cannot be extended. You may have one of the following reasons to make the class final:

  - **Security**: to prevent hackers from attempting to subvert a system by creating subclasses of a class and then substituting their class for the original.

  - **Design**: you may think that your class is perfect or that, conceptually, your class should have no subclasses

- If you mark a **method** with final, then they cannot be overridden.

  - You should make a method final if it has an implementation that should not be changed and its critical to the consistent state of the object.

  - One practical example is the Thread class. People writing multi-threaded programs will extend the Thread class, but the system implementers must prevent them from accidentally or deliberately redefining the individual method that do thread housekeeping (suspend(), resume(), …).

- A final method is also a cue to the compiler to inline the code. In Java all operations are dynamically dispatched by default. If you don't want to have dynamic dispatch, declare an operation as **final**.

Abstract Classes and Methods

- You specify abstract classes and abstract methods in Java with the abstract keyword.

- Once you have told Java that a class is abstract, you cannot create instances of the class. However, you can declare a variable of an abstract class.

```
public abstract class Figure {
        int x;
        public abstract draw();
        . . .
}

Figure b;
b  = new circle();
b.draw();
b.setDiameter(25);
```

- You can define abstract methods only in abstract classes

- An abstract class is not required to have an abstract method in it.

- Any class that has an abstract method in it or that does not provide an implementation for any abstract method declared in its superclass must be declared as an abstract class.

Pseudovariable “this”

- Pseudovariable **this** refers to the current object, the object whose method is being called.
- Can be used (not required) with variables and methods to make the reference to itself explicit (required in Smalltalk with self), and to return the object.
- Sometimes you need to disambiguate the instance variable name if one of the arguments to the method has the same name.

```
Class HSBColor {
        int hue, saturation, brightness;
        HSBColor(int hue, int saturation, int brightness) {
                this.hue = hue;
                this.saturarion = saturation;
                this.brightness = brightness;
         }
}
```

Pseudovariable “super”: refers to the superclass. Used to access a member of the superclass that was overriden in the class.

```
class Silly {
    boolean flag;
    void set() {
        flag = true;
      }
  }
```

```
class Sillier extends Silly {
    boolean flag;
        void set() {
            flag = false;
            super.set();
            System.out.println(flag);
            System.out.println(super.flag);
           }
```

Constructors

- Like C++, constructors in Java have the same name of the class, can be overloaded, and are automatically called when an object of the class is created.

- Constructors do not return a value, and have no return type.

- Constructors can have private, protected, package, and public access:

  - **Private**: no other class can instantiate the class as an object. The class can still contain public class methods, and those methods can construct an object and return it, but no one else can.

  - **Protected**: only subclasses can create instances of the class

  - **Package**: no one outside the package can create an instance

  - **Public**: anybody can create instances of the class

```
class Fruit {
        int grams;
        int calories;

        fruit() {
                grams = 55;
                calories = 0;
        }

        fruit(int g, int c) {
                grams = g;
                calories = c;
        }
```

How to Call other Constructors

- Whenever you want a constructor to hand arguments to another constructor in the same class, you can add a statement consisting of the **this** keyword followed by an argument list. The added statement must be the first statement in the constructor.

- Whenever you want a constructor to hand arguments to another constructor in the superclass, you can add a statement consisting of the **super** keyword followed by an argument list. The added statement must be the first statement in the constructor.

- You cannot arrange for explicit calls to more than one constructor

```
public class Student {
        public int id;
        public Student(int n) {
                id = n;
          }
        }


public class Grad extends Student {
        public int project;

        public Grad(int p) {
                project = p;
        }

        public Grad(int p, int n) {
                this(p);
                id = n;
        }
}
```

```
public class Student {
        public int id;
        public Student(int n) {
                id = n;
          }
        }


public class Grad extends Student {
        public int project;

        public Grad(int p, int n) {
                super(n);
                project = p;
        }
}
```

Constructors in C++

- C++ provides the alternative of using an initializer in a constructor:

```
Employee: Employee(String n, double s)
        :name(n), salary(s)
{ . . .}


Employee: Employee(String n, double s) {
        name = n;
        Salary = s;
}
```

- In the first example C++ constructs **name** directly as a copy of **n**.

- In the second case, **name** is constructed as an empty string first, and then it overwrites that string with **n**.

- Java object variables never contain any other object, just references to other objects.

- Object variables are initialized to **null** before entering the constructor body (essentially cost free).

- Java never call the default constructor automatically for data fields. You must explicitly call the default constructor to initialize it to a default value.

- A data field can be explicitly initialized in the class definition. They are carried out before the constructor executes.

```
Class Employee {
    private String name = "Default";
    private double salary = Math.random();
```

- In detail, what happens when an object is constructed in Java is:

1. All data fields are set to **zero**, **false**, or **null**.

2. If the constructor starts with the **this**, construct, then it recursively calls the other constructor.

3. The superclass constructor is invoked, either with parameters supplied in the **super** construct or with no parameters.

4. The data fields with initializers are set, in the order in which they appear in the class definition.

5. The constructor body is executed.

Example:

```
class Employee {
        public Employee() {
                name = new String();
                salary = 0;
        }
}


class Manager extends Employee {
        public Manager(String n, double s) {
                super(n,s);
        }
}
```

- If you don't call super in a subclass constructor, then the superclass will be constructed with its default constructor (error if it doesn't have one).

Common Error ???

```
class Employee {

        private String name;

        private double salary;

        public Employee(String n, double)
        {
                String name = n;
                double salary = s;
        }
}
```

Equality

- The default meaning of the equality operator ‘ == ’ is identity, or pointer equality.
- Two object variables are equal if they point to exactly the same object.
- It is not possible to overload operators, but you can define methods for equality:

```
Class Card {

    boolean equals(Card c)
    {
        return ( (suit == c.suit) &&
                                (rank == c.rank) );
    }
}
```

Double Dispatch: run-time resolution on two arguments.

- Polymorphism allows run-time selection of only one argument (receiver).
- What happens when a method is defined on arbitrary combinations of two polymorphic arguments??

  Ex: Suppose we have methods for the union and intersection of shapes (rects, ellipses,…)

```
class Rectangle  extends Shape {
        Shape intersect(Shape s) {… }
        . . . }
```

- A typical call would be:

```
r.intersect(s);
```

- If r is object of class Rectangle then by polymorphism it would respond to the message. However, we don't know what type of shape is s.
- We can take advantage of the fact that we know that r is a Rectangle, and call:

```
s.intersectRect(this);
```

- May require many methods to handle all possibilities.
- Difficult to maintain.
- Identical operations can be factored out.
- Convenient for small and stable collections of classes, when functionality can vary with both arguments.

Vectors: useful when you are unsure about how much information you need to be able to store (util package).

- addElement(n) // add to back end
- insertElement(n, ix) // insert & displace
- removeElementAt(ix) // remove & displace
- firstElement() // read front element
- lastElement() // read back element
- elementAt(ix) // retrieve element
- setElementAt(n, ix) //replace element
- size()

You can only store class instances in Vectors. Vectors cannot hold elements of primitive types.

Vectors of primitive types can be defined using **Wrappers**.

All elements of a Vector appear to be instances of **Object**. You must **cast** the element to use it as target to a method that belongs to a particular class.

```
Vector  v = new Vector();
Bird b = new Bird();
v.insertElementAt(b, 0);
( (Bird) v.firstElement()).fly();
```

Example:

Define a Queue, Stack using Vectors.

Wrappers

- Java has a classes for each primitive data type (int, long, float, double, char) that can hold a value of that type.

- This is useful since most of Java's utility functions require the use of objects. In addition a wrapper class can be used as a target of a method.

- Once a wrapper has been instantiated with a certain value, that value cannot be changed. However, you can always throw away that object and create an new one.

Example:

```
Integer mango;

int i = 38;

mango = new Integer(i);      // int to object

i = mango.intValue();        // object to int
```

Interfaces

Sometimes, you'd like classes from different parts of the class hierarchy to behave in similar ways.

Example: In a spreadsheet program that contains a tabular set of cells (containing one value) you want to put Strings, Floats, Dates, Integers, Equations, and so on.

To do so, all those classes would have to implement the same set of methods. How do we accomplish this???

1. Find the common ancestor and implement the methods there.

2. Create a class CellValue and make all other a subclass of it.

3. Use an Interface.

An Interface is a collection of method definitions (without implementations) and constant values.

- When a class claims to **implement** an interface, it is claiming that it provides a method implementation for all of those declared in the interface.

- You use interfaces to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Classes that implement an interface don't have to be hierarchically related.

Example: for the spreadsheet cell problem you could create an interface **CellAble** that defines the list of methods that a cell value must implement (toString, draw, toFloat,…).

Then the spreadsheet cells can contain any type of object that implements the interface.

Example:

```
public interface Collection {
        int MAX = 500;
        void add(Object obj);
        void delete(Object obj);
        Object find(Object obj);
        int currentCount();
}
```

- Interfaces have either public or package access.
- Constants declared are implicitly public, static, and final.
- Methods are implicitly public and abstract

- Interfaces can extend zero, one, or more interfaces.
- Cannot extend classes.
- Inherit all constants and methods from its superinterfaces unless it hides the constant with another of the same name or redeclares a method.
- Classes can implement zero, or more interfaces.

The method signatures of the class that implements the interface must match the corresponding method signatures declared in the interface.

Example:

```
class Queue implements Collection {
        . . .
        void add(Object obj)
                { . . . }

        void delete(Object obj)
                { . . . }

        Object find(Object obj)
                { . . . }

        int currentCount()
                { . . . }
}
```

Using an interface as a type

- When you define an interface you are in essence defining a new reference data type. Variables can be declared by the interface name.

- You can use interface names anywhere you use other data type names.

- Any object that implements the interface, regardless of where it exists within the class hierarchy can be assigned to a variable of that interface.

Example: Interface for spreadsheet program

```
interface CellAble {
        void draw();
        void toString();
        void toFloat();
}


class Row {

        private CellAble[] contents;
        . . .
        void setObjAt(CellAble c, int I) {
                . . .
        }
        . . .
}
```