What is Java?

❑ Result of Sun's efforts to remedy bad software engineering practices

❑ It is commonly thought of as a way to make Web pages "cool". It has evolved into much more. It is fast becoming a computing platform -- the base upon which software developers can build applications- traditional spreadsheets and word processors etc.

❑ Java, formerly known as oak, is an object-oriented programming language developed by Sun. It shares many superficial similarities with C, C++, and Objective C (for instance for loops have the same syntax in all four languages); but it is not based on any of those languages, nor have efforts been made to make it compatible with them.

❑ Java has a few things C++ doesn't have like garbage collection and multithreading; and discards some C++ features that had proven to be better in theory than in practice like multiple inheritance and operator overloading.

❑ Java was designed not only to be cross-platform in source form like C, but also in compiled binary form. (OK.. well.. not quite :)) Java is compiled to an intermediate byte-code which is interpreted on the fly by the Java interpreter. Thus to port Java programs to a new platform all that is really needed is to port the interpreter.

❑ Finally Java was designed to make it a lot easier to write bugfree code. Shipping C code has, on average, one bug per 55 lines of code. About half of these bugs are related to memory allocation and deallocation. Thus Java has a number of features to make bugs less common: Strong typing, Small language, easy to read, no architecture dependent constructs, object oriented, concurrency support.

Applications and Applets:

Application: Java program that executes independently of any browser.

Applet: Java program to be included in HTML pages and executed in a Java-compatible browser.

The "Hello World" Application

```
/* File HelloWorldApp.java */

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

Compile the source file

```
% javac HelloWorldApp.java
```

Run the application

```
% java HelloWorldApp
```


The "Hello World" Applet

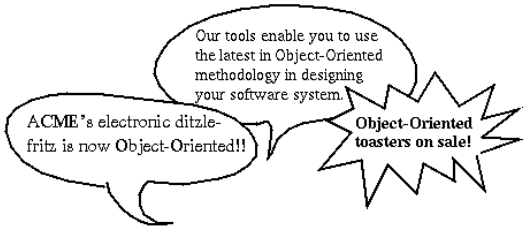
```
/* file: HelloWorld.java */
import java.awt.Graphics;
      public class HelloWorld extends java.applet.Applet {
            public void init() {
                  resize(150,25);
            }
            public void paint(Graphics g) {
            g.drawString("Hello world!", 50, 25);
      }
}
```

Compile the file to a class file using javac.

Create an HTML file that includes the applet

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>
Here is the output of my program:
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```


Writing Java Programs: Object-Oriented Programming Concepts

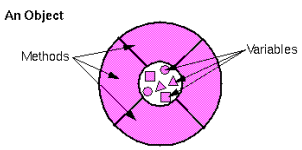


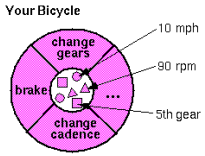
Is object oriented programming the next best thing to white bread?!
(who likes bread anyway :)) Maybe not, but it sure does a whole lot of good for how we write software. Let us briefly venture to see how.

Object Oriented Programming: Objects, Messages and Classes.

What is an Object?

Objects are software bundles of data and related methods.





The Benefit of Encapsulation

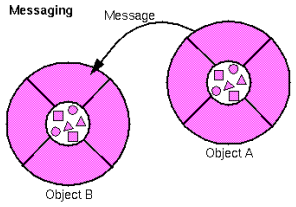
Modularity--the source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else and it will still work.

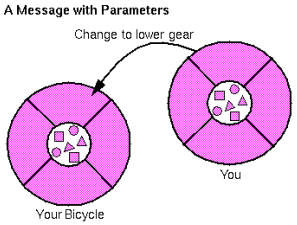
Information hiding--an object has a public interface which other objects can use to communicate with it. But the object can maintain private information and methods that it can change at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike in order to use it.

Error Control -- You can prevent improper operations on data.

What are Messages?

Software objects interact and communicate with each other via messages. When object A wants object B to perform one of its methods, object A sends a message to object B.





Examples:

Object (you) pedals (message) Object (bicycle)
Object (you) commands (kneel) Object (dog).

Note that you cant make a bicycle kneel or pedal a dog. Also that objects can be across processes or even machines.

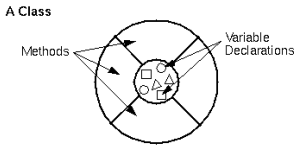
Three components comprise a message:

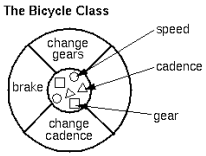
1.the object to whom the message is addressed (bicycle)
2.the name of the method to perform (change gears)
3.any parameters needed by the method.

What are Classes?

In the real world, you often have many objects of the same kind. For example, your bicycle is really just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle is an instance of the class of objects knowns as bicycles. All bicycles have some states (current gear, current cadence, two wheels) and behaviours (change gears, brake) in common.

A class is a template or prototype that defines the variables and the methods common to all objects of a certain kind.





The values for the variables are provided by each instance of the class. So, after you've created the bicycle class, you must instantiate it (create an instance of it) to use it. When you create an instance from a class, the variables declared by the class are allocated in memory. Then you can use the instance's methods to assign values to the variables. Instances of the same class share method implementations.

What is Inheritance?

Generally speaking, objects are defined in terms of classes. Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, race bikes and tandems are all different kinds of bicycles. In object-oriented terminology, mountain bikes, race bikes and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, race bikes and tandems.

Each subclass *inherits* state (in the form of variable declarations) from the superclass. Mountain bikes, race bikes and tandems share some states: cadence, speed and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, race bikes and tandems share some behaviours: braking and changing pedaling speed.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could actually use those new gears.

The Benefit of Inheritance

Subclasses provide specialized behaviours from the basis of common elements provided by the superclass.

Program development issues: One person defines the superclasses and other (lesser programmers :)) can fill in the rest of the class hierarchy.

The Life Cycle of an Object

The typical life-cycle of an object is 1) creation, 2) use, and 3) destruction. After an object has been created, any other object (that has access) can use its methods, or, inspect its variables. An object remains “alive” until no other objects are using it. Once an object has completed its useful life, you don’t have to do anything--the Java runtime system will clean it up for you!

Creating an Object

In Java, you create an object by creating an instance of a class or, in other words, instantiating a class. To create a new object, use Java’s new operator plus a constructor for the type of object that you want to create.

```
new Constructor(...)
```

Constructors are special methods provided by every Java class allowing programmers to create and initialize objects of that type.

Constructors have the same name as the class and, because Java supports method name overloading, a class can have any number of constructors. The number and type of arguments determine which constructor would be used.

Creating Your Own Class

A class is a template used to create many different objects. It is convenient to think of a class as a data type. Indeed, objects created from the same class are often spoken of as being of the same type. In your Java programs you can use classes provided by other programmers, such as the classes provided by the Java development environment. Or you can write your own.

The implementation of a class is comprised of 2 components: the class declaration and the class body.

```
class declaration {
   . . .
   class body
   . . .
}
```

Declaring a Class

In general, a class declaration look like this:

```
[ modifiers ] class ClassName [ extends SuperClassName ]
                                          [ implements InterfaceNames ] {
   . . .
   class body
   . . .
}
```


A simple class declaration

```
class ImaginaryNumber {
   . . .
}
```

A class declaration defines the following aspects of the class:

modifiers declares whether or not the class is abstract, final or public
ClassName sets the name of the class you are declaring
SuperClassName is the name of ClassName's superclass
InterfaceNames is a list of all of the interfaces implemented by the *ClassName*

Our simple ImaginaryNumber class declaration assumes that

the Object class is the super class of ImaginaryNumber
ImaginaryNumber implements no interfaces
ImaginaryNumber is not abstract, not final, and is not public.

The class of circles

```
public class Circle {
   public double x, y; // coordinates of the center
   public double r;    // radius of the circle

   public double circumference() {
        return 2 * 3.14 * r;
   }

   public double area () {
        return 3.14 * r * r;
   }
}
```

To declare an instance of this class, use

```
Circle c;
```

This only declares a reference to the class (it does not allocate memory for the class)
To instantiate the class, try

```
c = new Circle ();
```

Using the class Circle

Try the following:

```
Circle c = new Circle();
c.x = 2.0;
c.y = 2.0;
c.r = 1.0;
// at this point we have defined a circle of radius 1
// let us try to use one of the methods
a = c.area();
// This computes the area of the circle c and assigns it to a
```

The arguments to the method are passed implicitly. In fact, they are passed as an object called this. To test this, change the area method to:

```
public double area (){
      return 3.14 * this.r * this.r;
}
```


Constructors

The call c = new Circle(); is in fact a call to a constructor that initializes the class. Since we have not defined a constructor in the lass, the system uses a default constructor to initialize the instance. Here is an example of a constructor;

```
public class Circle {
        double x, y, r;
        public Circle () {
              x = 0.; y = 0.; r = 0.;
        }

        public Circle(double x, double y, double r) {
              this.x = x; this.y = y; this.r = r;
        }

        public Circle (double r) {
              x = 0.; y = 0.; this.r = r;
        }

        public Circle (Circle c) {
              x = c.x; y = c.y; r = c.r;
        }
}
```

Try each of the following now

```
c = new Circle(); c = new Circle(1., 2., 3.); c = new Circle(2.); c = new Circle(d);
```


Superclasses

The Object class sits at the top of the class hierarchy tree in the Java development environment. Every class, whether written by you, by the Java development team, or by someone else, in the Java system is a descendent (whether direct or indirect) of the Object class.

To specify an object's superclass explicitly, put the keyword extends followed by the superclass name directly after the name of the class that you are declaring. For example:

```
class ImaginaryNumber extends Number {
   . . .
}
```

Interfaces

An interface declares a set of methods and constants without specifying the implementation for any of the methods. When a class claims to implement an interface, it's claiming to provide implementations for all of the methods declared in the interface.

For example, imagine an interface named Arithmetic that defines methods named add(), subtract() and so on. Our ImaginaryNumber class can declare that it implements the Arithmetic interface, thereby guaranteeing that it provides implemenations for the add(), subtract() and other methods declared by the Arithmetic interface.

Interfaces (continued)

```
class ImaginaryNumber extends Number implements Arithmetic {
   . . .
   public Number add(Number) {
      . . .
   }
   public Number subtract(Number) {
      . . .
   }
}
```

Abstract Classes: An abstract class is a class that has at least one abstract method in it. An abstract method is a method that has no implementation. Use an abstract class when your parent class knows that all of its subclasses will implement a certain method but each subclass will implement it differently.

Final Classes: A final class can have no subclasses. You would a class final to guarantee that other objects and methods using your class got exactly what they asked for. For example, the String class in the java.lang package is a final class.

Public Classes: Use public to declare that the class can be used by objects outside the current package.

```
public final class ImaginaryNumber extends Number implements Arithmetic {
                . . .
}
```

Writing a Method

A method has a name and a body.

```
return_type method_name() {
    . . .
    method body
    . . .
}
```

For example:

```
boolean isEmpty() {
    . . .
}
```

But what is empty ?? We need to associate this method with a class. The combination of a class and method give meaning to a method. For example, isEmpty may be associated with a stack class.

```
class declaration {
    . . .
    member variable declarations
    . . .
    method declarations
    . . .
}
```


Writing Methods.

Let's put the isEmpty() method in an appropriate class:

```
class stack {
    . . .
    boolean isEmpty() {
        . . .
    }
}
```

Returning a Value from a Method

Methods can return simple data types (integer numbers, floating point numbers, boolean values, and characters (a single character, that is)) or complex data types (classes, interfaces, and arrays)

```
class Stack {
    static final int STACK_EMPTY = -1;
    Object stackelements[];
    int topelement = STACK_EMPTY;
    . . .
    boolean isEmpty() {
        if (topelement == STACK_EMPTY)
            return true;
        else
            return false;
    }
}
```


Returning a Value from a Method

The following method returns a complex data type: an object.

```
class Stack {
    static final int STACK_EMPTY = -1;
    Object stackelements[];
    int topelement = STACK_EMPTY;
    . . .
    Object pop() {
        if (topelement == STACK_EMPTY)
            return null;
        else {
            return stackelements[topelement--];
        }
    }
}
```

Non-void declared methods must have a return statement. The return types should match.

More on writing methods:

By default, when you declare a method within a class that method is an instance method. All instances of the class share the same implementation of an instance method.

```
class AnIntegerNamedX {
    int x;
}
```

Every time you instantiate AnIntegerNamedX, you create an instance of AnIntegerNamedX and each instance of AnIntegerNamedX gets its own copy of x. To access a particular x, you must access it through the object with which it is associated.

Now, let's make x private, and create two public methods within AnIntegerNamedX which allows other objects to set and query the value of x.

```
class AnIntegerNamedX {
    private int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

More on writing methods:

Now the following code snippet creates two different instaces of type AnIntegerNamedX, sets their x values to different values using the new setX() method, and then prints out the values.

```
. . .
AnIntegerNamedX myX = new AnIntegerNamedX();
AnIntegerNamedX anotherX = new AnIntegerNamedX();
myX.setX(1);
anotherX.setX(2);
System.out.println("myX.x = " + myX.x());
System.out.println("anotherX.x = " + anotherX.x());
. . .
```

The output produced by this code snippet is

```
myX.x = 1
anotherX.x = 2
```


Declaring a Member Variable

In Java, there are two different kinds of variables: variables that are associated with an object or class (called member variables) and those that are not associated with an object or class but are used locally within methods and other code blocks (local variables, parameters, etc.).

There are two different types of member variables: class variables and instance variables. A class variable occurs once per class regardless of the number of instances created of that class. The system allocates memory for class variables the first time it encounters the class. An instance variable occurs once per instance of a class.

```
class declaration {
   . . .
   member variable declarations
   . . .
   method declarations
   . . .
}
```

For example:

```
class IntegerClass {
   int anInteger;
}
```


Using Class Variables:

To specify that a variable is a class variable, use the keyword static. For example, let's change the AnIntegerNamedX class such that its x variable is now a class variable and that it's x() and setX() methods are now class methods:

```
class AnIntegerNamedX {
   static private int x;
   static public int x() {
      return x;
   }
   static public void setX(int newX) {
      x = newX;
   }
}
```

Now the exact same code snippet from before that creates two instances of AnIntegerNamedX, sets their x values, and then prints the x values produces this, different, output.

```
myX.x = 2
anotherX.x = 2
```

You can use class variables for storing the number of circles in the circle class. This is Java's version of global variables.

Declaring a member variable:

A member variable declaration looks like this:

```
[accessSpecifier] [static] [final] [transient] [volatile] type variablename
```

The items between [and] are optional. Italic items are to be replaced by keywords or names.

A variable declaration defines the following aspects of the variable:

accessSpecifier defines which other classes have access to the variable

static indicates that the variable is a class member variable as opposed to an instance member variable

final indicates that the variable is a constant

```
final double AVOGADRO = 6.023e23;
```

transient variables are not part of the object's persistent state

```
transient int hobo;
```

volatile means that the variable is modified asynchronously

```
volatile int counter;
```

Declaring class methods

Just like variables, there are instance methods and class methods. Class methods are declared static. Let us explore these using an example:

```
public class Circle {
      double x, y, r;
      // is point (a,b) inside the circle
      public boolean isInside (double a, double b)
      {
            double dx = a - x;
            double dy = b - y;
            double distance = Math.sqrt(dx * dx + dy * dy);
            if (distance < r) return true;
            else return false;
      }
}
```

Here, Math is a class defined by java and Math.sqrt is a method that has been defined as a class method (i.e. as static). To call this method, we do not have to go through an instance, rather we can go through the class itself. These methods are called class methods.

Controlling Access to a Class's Variables

The Java language supports five distinct access levels for variables: private, private protected, protected, public, and, if left unspecified, "friendly". The following chart shows the access level permitted by each specifier.

Specifier	class	sub-class	pack-age	world
private	X			
private protected	X	X		
protected	X	X*	X	
public	X	X	X	X
friendly	X		X	

The first column indicates whether or not the class itself has access to the variable defined by the access specifier. The second column indicates whether or not subclasses of the class (regardless of which package they are in) have access to the variable. The third column indicates whether or not classes in the same package (regardless of their parentage) as the class have access to the variable. And finally, the fourth column indicates that all classes have access to the variable.

Private

A private variable is only accessible to the class in which it is defined. To declare a private variable, use the keyword private. For example, the following class defines one private variable within it:

```
class Alpha {
   private int iamprivate;
}
```

Objects of type Alpha can inspect or modify the iamprivate variable, but objects of other types cannot. For example, the following class, regardless of which package it is in or its parentage, cannot access the iamprivate variable within the Alpha class.

```
class Beta {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprivate = 10;      // illegal
   }
}
```

You can tell when one of your classes is attempting to access a variable to which it does not have access--the compiler will print an error message similar to the following and refuse to compile your program.

```
Beta.java:9: Variable iamprivate in class Alpha not accessible from class
Beta.
      a.iamprivate = 10;     // illegal
       ^
1 error
```


Private Protected

The next most restrictive access specified is private protected. This access level includes the same access level as private plus allows any of the class's subclasses to access the variable. To declare a private protected variable, use the keywords private protected. For example, the following class defines one one private protected variable within it:

```
class Alpha {
   private protected int iamprivateprotected;
}
```

Objects of type Alpha can inspect or modify the iamprivateprotected variable. In addition, subclasses of Alpha also have access to iamprivateprotected. For instance, this subclass of Alpha can assign its iamprivateprotected variable to that of another Alpha object.

```
class Beta extends Alpha {
   void modifyVariable(Alpha a) {
      a.iamprivateprotected = this.iamprivateprotected;      // legal
   }
}
```

Protected

The next access level specifier is protected which allows the class itself, subclasses (with a caveat), and all classes in the same package to access the variable. To declare a protected variable, use the keyword protected. For example, take this version of the Alpha class which is now declared to be within a package named “Greek” and which has a single protected variable declared within it.

```
package Greek;

class Alpha {
   protected int iamprotected;
}
```

Now, suppose that the class, Beta, was also declared to be a member of the Greek package. The Beta class can legally access the iamprotected variable declared within the Alpha class.

```
package Greek;

class Beta {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprotected = 10;    // legal
   }
}
```


Public

Now for the easiest access specifier--public. To declare a public variable, use the keyword public. For example,

```
class Alpha {
   public int iampublic;
}
```

Any class, in any package, has access to a class’s public variables. For example, this version of the Beta class

```
class Beta {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iampublic = 10;       // legal
   }
}
```

can legally inspect and modify the iampublic variable in the Alpha class.

Friendly

And finally, the last access level is what you get if you don’t explicitly set a variable’s access to one of the other levels. For example, this version of the Alpha class declares a single “friendly” variable and lives within the Greek package.

```
package Greek;

class Alpha {
   int iamfriendly;
}
```

The Alpha class has access to iamfriendly. In addition, all the classes declared within the same package as Alpha also have access to iamfriendly. For example, suppose that both Alpha and Beta were declared as part of the Greek package, then this Beta class

```
package Greek;

class Beta {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprotected = 10;    // legal
   }
}
```

could legally access iamfriendly.

A Variable’s Type

Keyword	Size	Description
(integer types)		
byte	8-bit	Byte-length integer
short	16-bit	Short integer
int	32-bit	Integer
long	64-bit	Long integer
(floating-point types)		
float	32-bit	single-precision floating point
double	64-bit	double-precision floating point
(other types)		
char	16-bit	a single Unicode character
boolean	N/A	a Boolean