

# Introduction To Java

Java Server Technologies

Bob Tarr

## What Is Java?

- New object-oriented programming (OOP) language developed by SUN Microsystems
- Similar to C and C++, except without some of the confusing, poorly understood features of C++
- Extensive networking facilities
- Extensive set of APIs for GUIs, distributed computing, 2D/3D graphics, mail, and others
- Portable: Write Once, Run Anywhere
- Multithreading support built into the language

Java Server Technologies

Introduction To Java  
2

Bob Tarr

## Java Features

- Automatic garbage collection
  - ⇒ No manual memory allocation and deallocation
  - ⇒ Never have to worry about memory leaks
- No pointers or pointer arithmetic
  - ⇒ No off-by-one bugs
- Arrays are first-class objects
  - ⇒ Array bounds are always checked
- Multiple inheritance replaced by interfaces
  - ⇒ Eliminates complexities of multiple inheritance

*Improves software reliability*  
*Increases programmer productivity*

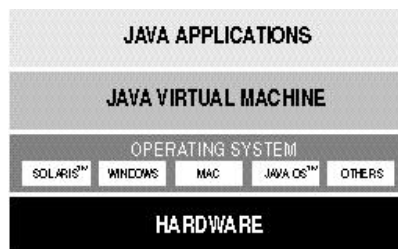
## Features Removed From C/C++

- No typedefs, defines or preprocessor
- No header files
- No structures or unions
- No enums
- No functions - only methods in classes
- No multiple inheritance
- No goto
- No operator overloading (except “+” for string concatenation)
- No automatic type conversions (except for primitive types)
- No pointers

## Java Virtual Machine

- Java is compiled into bytecodes
- Bytecodes are high-level, machine-independent instructions for a hypothetical machine, the Java Virtual Machine (JVM)
- The Java run-time system provides the JVM
- The JVM interprets the bytecodes during program execution
- Since the bytecodes are interpreted, the performance of Java programs slower than comparable C/C++ programs
- But the JVM is continually being improved and new techniques are achieving speeds comparable to native C++ code

## Java Virtual Machine



- All Java programs run on top of the JVM
- The JVM was first implemented inside Web browsers, but is now available on a wide variety of platforms
- The JVM interprets the bytecodes defined in a machine-independent binary file format called a *class* file

## Types Of Java Programs

- Application
  - ⇒ Standalone Java program that can run independent of any Web browser
- Applet
  - ⇒ Java program that runs within a Java-enabled Web browser
- Servlet
  - ⇒ Java software that is loaded into a Web server to provide additional server functionality ala CGI programs

## The Hello World Program

- Create source file: Hello.java

```
public class Hello {
    public static void main (String args[]) {
        System.out.println("Hello World!");
    }
}
```
- Note that the name of the file is the name of the public class with a .java extension added
- Compile: javac Hello.java
  - ⇒ Produces the class file Hello.class
- Run: java Hello
  - ⇒ Starts up the JVM
  - ⇒ Note that the .class extension is *not* specified

## Basic Java Language Elements

- Primitive Types

- ⇒ byte (1 byte) -128 to +127
- ⇒ short (2 bytes) -32,768 to 32,767
- ⇒ int (4 bytes) -2.15E+9 to +2.15E+9
- ⇒ long (8 bytes) -4.61E+18 to +4.61E+18
- ⇒ float (4 bytes) 1.0E-38 to 1.0E+38
- ⇒ double (8 bytes) -1.0E-308 to 1.0E+308
- ⇒ char (2 bytes) 0 to 0xffff (Unicode)
- ⇒ boolean (1 byte) *true* and *false*
- ⇒ All numeric types are sign extended

## Basic Java Language Elements

- Operators

- ⇒ Similar to C++
- ⇒ Differences between C++ and Java Operators:
  - + is used to concatenate strings
  - instanceof returns true or false depending on whether the left side object is an instance of the right side object
  - >>> shifts bits right, filling with zeros

- Decision Constructs

- ⇒ if-else (test expression must resolve into a boolean)
- ⇒ switch (switch expression must resolve to an int, short, byte or char)

- Loops

- ⇒ Same as C/C++
- ⇒ while, do-while
- ⇒ for

### Some Java OO Terminology

- *Class* - collection of data (attributes) and methods that operate on that data
- *Member* - either an attribute or a method of a class
- *Public Member* - member which is accessible by any method in any class
- *Private Member* - member which is accessible only by methods defined within the class
- *Public Class* - class that is visible everywhere and can be used by any method in any class
- *Object* - instance of a class
- *Object Instantiation* - the creation of a new object

### Some Java OO Terminology

- *Constructor* - method which performs object initialization (*not* creation!)
- *Object Reference* - variable that holds a reference to (really the memory address of) an object
- *Instance Variable* - attribute for which each object (instance) has its own copy
- *Class Variable* - attribute for which there is only one copy for the class. Each object (instance) shares this copy. Also called a *static variable*

## Some Java OO Terminology

- *Instance Method* - method which operates on the attributes of an object (instance)
- *Class Method* - method which does not operate on a particular object, but performs some utility function or operates on static variables. Also called a *static method*.
- *Method Signature* - the number, type and order of arguments of a method
- *Method Overloading* - defining a method with the same name but different signature as another method in the same class

## Simple Java Example: Point

```
/**
 * Class Point implements a geometric point.
 * @author Bob Tarr
 */
public class Point {

    private int x; // X Coordinate
    private int y; // Y Coordinate

    /**
     * Creates a new Point with coordinates 0,0.
     */
    public Point() {
        x = 0;
        y = 0;
        System.out.println("Point() constructor: " + this);
    }
}
```

## Simple Java Example: Point (Continued)

```
/**
 * Creates a new Point with the specified coordinates.
 * @param x The x coordinate.
 * @param y The y coordinate.
 */
public Point(int x, int y) {
    this.x = x;
    this.y = y;
    System.out.println("Point(int,int) constructor: " + this);
}

/**
 * Returns the x coordinate.
 * @return The x coordinate.
 */
public int getX() {return x;}
```

## Simple Java Example: Point (Continued)

```
/**
 * Returns the y coordinate.
 * @return The y coordinate.
 */
public int getY() {return y;}

/**
 * Sets the x coordinate.
 * @param x The x coordinate.
 */
public void setX(int x) {this.x = x;}

/**
 * Sets the y coordinate.
 * @param y The y coordinate.
 */
public void setY(int y) {this.y = y;}
```

## Simple Java Example: Point (Continued)

```
/**
 * Converts a Point to a String.
 * @return The Point as a String.
 */
public String toString() {
    return "[" + x + "," + y + "]";
}
}
```

## Test Program For Point

- Test program for the Point class:

```
// Test program for the Point class.
public class TestPoint {
    public static void main(String args[]) {
        // Create some Point objects.
        Point p1 = new Point();
        Point p2 = null;
        p2 = new Point(5,10);

        // Test the accessors and mutators.
        p1.setX(22);
        System.out.println("P1 is now: " + p1);
        p2.setY(13);
        System.out.println("P2 is now: " + p2);
    }
}
```

## Test Program For Point (Continued)

- Test program output:

```
Point() constructor: [0,0]
Point(int,int) constructor: [5,10]
P1 is now: [22,0]
P2 is now: [5,13]
```

## Arrays

- Arrays are objects in Java
- Creating an array involves three steps: declaration, creation and initialization
- Declaration:

```
Point data[]; // The variable data can hold a reference
              // to an array of Points
Point[] data; // Same thing!
```

- Creation:

```
data = new Point[10]; // Now the variable data refers to the
                      // array of 10 elements that can refer
                      // to a Point object. All the references
                      // in the array are null.
```

## Arrays

- Initialization:

```
data[0] = new Point(4, 5); // First array element initialized.  
                        // It is now referring to the new  
                        // Point object.
```

- Declaration, creation and initialization can be combined:

```
int[] values = {1,7,5,8,9};  
Point[] points = {new Point(4,5), new Point(1,-3)};
```

## Exceptions

- *Exception* - a signal that an error or special condition has occurred
- *Throw an exception* - to signal the error or special condition
- *Catch an exception* - to handle the error or special condition
- Exceptions propagate up the lexical block structure of the Java program until they are caught and handled
- If an exception is not handled, the Java interpreter will print an error message and stack trace and then exit

## Exceptions

- Exceptions handling is done within a try/catch block:

```
try {
    // Try this code.  If it generates an exception,
    // we'll handle it in a catch block.
}
catch (Exception1 e1) {
    // Handle exception type Exception1.
}
catch (Exception2 e2) {
    // Handle exception type Exception2.
}
finally {
    // Always execute this code.
}
```

## Exceptions

- All exceptions in Java are objects derived from the Exception class
- Exceptions are of two types:
  - ⇒ Unchecked exceptions: These are exceptions that commonly occur, such as divide by zero. (They are instances of RuntimeException, a subclass of Exception).
  - ⇒ Checked exceptions: These are less common exceptions, such as an I/O error.
- Checked exceptions must either be caught or specified, else a compiler error will result
- A throws clause is used to indicate that the method may throw an exception up the call stack:

```
public void someMethod() throws IOException { ... }
```

## Inheritance

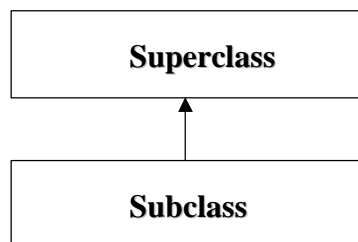
- Java supports single inheritance using the *extends* keyword:

```
public class B extends A
```

- The extended class is called a *subclass* of the class it extends
- The class that is extended is called its *superclass* or base class
- All classes implicitly extend the Object class
- A subclass can *override* a method in its superclass by providing a new definition for a method with the same name, return type and signature
- All method invocations in Java are *polymorphic*. The method called depends on the type of the object referred to by its object reference and not on the type of the object reference itself.

## Inheritance

- A *protected member* of a class can be accessed by any method in the same class or a subclass. (It can also be access by any method in a class in the same package which will be described later.)
- UML Notation:



## Inheritance Example

- Here's the superclass:

```
public class A {  
  
    protected int aData;  
  
    public A(int aData) {this.aData = aData;}  
  
    public A() {aData = 0;}  
  
    protected void f() {  
        System.out.println("This is A's f");  
    }  
  
}
```

## Inheritance Example (Continued)

- Here's the subclass:

```
public class B extends A {  
  
    protected int bData;  
  
    public B(int bData) {this.bData = bData;}  
  
    public B() {this(0);}  
  
    protected void f() {  
        System.out.println("This is B's f");  
    }  
  
    protected void g() {  
        System.out.println("This is B's g");  
    }  
  
}
```

## Inheritance Example (Continued)

- Here's the test program:

```
public class ABTest {

    public static void main(String[] argv) {

        //Polymorphism
        A a = new A();
        B b = new B();
        a.f();           // Invokes A's f()
        b.f();           // Invokes B's f()
        A a1 = b;
        a1.f();          // Invokes B's f()

        // Up Casting
        A a2 = (A) b;    // Ok
    }
}
```

## Inheritance Example (Continued)

```
// Down Casting
//B b1 = a;           // Illegal at compile time,
//                   // explicit cast needed
//B b2 = (B) a;       // Ok at compile time,
//                   // exception at run time

// Other stuff
int i = a.aData;     // Ok, same package
//i = a.bData;       // Illegal at compile time,
//                   // bData not defined in A
//a.g();             // Illegal at compile time,
//                   // g() not found in A
}

}
```

## Constructor Chaining

- Java always invokes a superclass constructor when a subclass object is created (since the superclass object is “part of” the subclass object)
- You can explicitly call a superclass constructor using a call to `super(...)` as the first line of a subclass constructor:

```
public B(int bData) {  
    super(); // Explicitly call our superclass constructor  
    this.bData = bData;  
}
```

- If you do not explicitly invoke a superclass constructor, then the no-arg superclass constructor is implicitly called for you. That is, Java inserts the call to `super()` for you automatically.

## Constructor Chaining

- What? You don't have a no-arg superclass constructor? That's ok, provided you have *no* superclass constructors, in which case the default no-arg constructor for a class is supplied for you. (But if you have superclass constructors defined, and do not have a no-arg one, you'll get a compiler error when Java tries to insert the call to `super()` in the subclass constructor.)
- The default no-arg constructor supplied by Java does just one thing - it makes a call to the no-arg superclass constructor!
- One exception: If the first line of a constructor uses the `this(...)` syntax to invoke another constructor of the class, then Java does not automatically insert the call to `super()` in the constructor:

```
public B() {  
    this(0); // Call to super() not automatically inserted here.  
}
```

## Abstract Methods And Classes

- An *abstract method* has no body
- It is like a pure virtual function in C++.
- The abstract method is expected to be overridden in a subclass with an actual implementation
- Any class with an abstract method is an *abstract class* and must be declared abstract
- An abstract class can not be instantiated
- If a subclass of an abstract class does not provide implementations for all of the abstract methods of its superclass, then the subclass itself is abstract

## Abstract Class Example

```
// Class Shape is an abstract base class for a geometric shape.
public abstract class Shape {
    public abstract double area();
}

// Class Rectangle is a concrete implementation of a Shape.
public class Rectangle extends Shape {

    protected double w;
    protected double h;

    public Rectangle(double w, double h) {
        this.w = w;
        this.h = h;
    }

    public double area() { return (w * h); }
}
```

## Interfaces

- In OO terminology, an *interface* is some subset of the public methods of a class. The *implementation* of a class is the code that makes up those methods.
- In Java an *interface* is just a specification of a set of abstract methods
- A class that implements the interface must provide an implementation for all of the abstract methods in the interface
- A class can implement many interfaces, but a class can only extend one class
- So a Java interface expressly separates the idea of an OO interface from its implementation

## Interface Example

```
// Interface Drawable provides the specification for a drawable
// graphics object.
public interface Drawable {
    public void Draw();
}

// Class DrawableRectangle implements the Drawable interface.
public class DrawableRectangle
    extends Rectangle
    implements Drawable {

    // Other code here.

    public void Draw() {
        // Body of Draw()
    }
}
```

## Differences Between Interfaces and Abstract Classes

- A class can implement more than one interface, but an abstract class can only subclass one class
- An abstract class can have non-abstract methods. All methods of an interface are implicitly (or explicitly) abstract.
- An abstract class can declare instance variables; an interface can not
- An abstract class can have a user-defined constructor; an interface has no constructors
- Every method of an interface is implicitly (or explicitly) public. An abstract class can have non-public methods.

## Why Are Interfaces Important?

- An object's type essentially refers to the OO interface of its class
- So, in Java, an object of a class that implements several interfaces has many types
- And objects from many different classes can have the same type
- This allows us to write methods that can work on objects from many different classes which *can even be in different inheritance hierarchies*:

```
public void renderScreen(Drawable d) {  
    // Render this Drawable on the screen.  
    // It does not matter whether this is DrawableRectangle,  
    // DrawableCircle, etc. Since the object is a Drawable, it  
    // MUST implement the Draw method.  
    d.Draw();  
}
```

## Packages

- Java classes can be grouped together into a *package*
- Packages have several advantages:
  - ⇒ related classes can be grouped together
  - ⇒ class names and member names need not be unique across the entire program
  - ⇒ members can be accessible only to methods of classes in the same package
- The package statement must appear as the first statement of the Java source file:

```
package BT.Tools.Graphics;
```

- If no package statement is present, the code is made part of the unnamed default package

## Packages

- A fully qualified Java name for a class is:
  - ⇒ <Package Name>.<Class Name>
  - ⇒ For example, BT.Tools.Graphics.Point
- Class files must be stored in a directory that has the same components of the package name for the class
  - ⇒ For example, the class BT.Tools.Graphics.Point must be in the BT/Tools/Graphics/Point.class file
  - ⇒ This filename is interpreted relative to one of the directories specified in the CLASSPATH environment variable

## Packages

- The CLASSPATH environment variable tells the Java interpreter where to look for user-defined classes. CLASSPATH is a colon-separated list of directories to search or the names of "zip" or "jar" files that contain the classes:
  - ⇒ For example, setenv CLASSPATH  
./home/bt/java/usr/local/comms/classes.zip
  - ⇒ Given the above CLASSPATH, if the Point.class file is in  
/home/bt/java/BT/Tools/Graphics, it will be successfully found by the Java run-time system

## The Import Statement

- The import statement allows the use of abbreviated class names in a Java source file
- Classes are always available via their fully-qualified names:

```
BT.Tools.Graphics.Point p = new BT.Tools.Graphics.Point();
```
- The import statement does not "read in" the class or "include" it; it just saves typing:

```
import BT.Tools.Graphics.Point;
Point p = new Point();
```
- All of the classes of a package can be imported at one time using this form of the import statement:

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

This imports all of the classes in the java.util package.

## Visibility Modifiers

- We've already seen that a class member can be modified with the public, private or protected keywords
- If none of these modifiers are used, the member has the default visibility or "package" visibility
- A *package member* is only accessible from within the class that defines it or a class in the same package
- Here's the definitive chart!

ACCESSIBLE TO:	MEMBER VISIBILITY			
	public	protected	package	private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass in different package	Yes	No	No	No

Table from *Java In A Nutshell, 2<sup>nd</sup> Edition* by David Flanagan

## Inner Classes

- Inner classes were added to the Java language in Java 1.1
- There are now five different types of Java classes and two different types of Java interfaces
- Top-level classes and interfaces
  - ⇒ *Package member class (or interface)*
    - Ordinary class (or interface) that is a direct member of a package
    - The original, familiar Java 1.0 class (or interface)
  - ⇒ *Nested top-level class (or interface)*
    - A class (or interface) declared static within another top-level class (or interface)
    - Can only access the static members of its containing class
    - Useful for helper classes (and interfaces) and provide a convenient way to group related classes (or interfaces)

## Inner Classes (Continued)

- Inner classes

⇒ *Member class*

- A class defined as a member of another class
- Can not be declared static
- Can not have any static members
- Can access all members (even private) of its containing class
- Also useful for helper classes

⇒ *Local class*

- Class defined inside a block of code
- Is visible only within the enclosing block
- Analogous to a local variable
- Can access all members (even private) of its enclosing class
- Similar in use to a member class, but can be put close to the location in the code where it is actually used, thus improving readability

## Inner Classes (Continued)

- Inner classes

⇒ *Anonymous class*

- A local class which is defined and instantiated in one statement
- Does not have a name!

- Inner classes are frequently used to implement the event listener objects required by the Abstract Window Toolkit (AWT) or Swing Java Foundation Classes GUI components

## Member Class Example

```
// Member Class example.
public class ButtonDemo {
    public ButtonDemo() {
        // Create a button.
        Button button = new Button("Press me");
        // Register an ActionListener for the button.
        button.addActionListener(new ButtonActionHandler());
        ...
    }
    ...
    // Somewhere later in the file we have this member class which
    // defines the required ActionListener.
    class ButtonActionHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println("You pressed me, you pressed me!");
        }
    }
}
}
```

## Local Class Example

```
// Local Class example.
public class ButtonDemo {
    public ButtonDemo() {
        // Create a button.
        Button button = new Button("Press me");
        // Register an ActionListener for the button.
        button.addActionListener(new ButtonActionHandler());

        // Let's put the definition of the required ActionListener right
        // here as a local class. That way, it is much closer in the
        // source file to its actual use.
        class ButtonActionHandler implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                System.out.println("You pressed me, you pressed me!");
            }
        }
    }
}
}
```

## Anonymous Class Example

```
// Anonymous Class example.
public class ButtonDemo {
    public ButtonDemo() {
        // Create a button.
        Button button = new Button("Press me");

        // Instantiate an anonymous inner class that acts as the
        // ActionListener for the button.
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("You pressed me, you pressed me!");
            }
        });
        ...
    }
    ...
}
```

## Anonymous vs Local

- Which one should you use? An anonymous class or a local class?
- It's a matter of your own personal style
- But....
- Prefer an anonymous class if
  - ⇒ The class is very small
  - ⇒ Only one instance of the class is needed
  - ⇒ The class is to be used right after it is defined
  - ⇒ Naming the class does not make your code any easier to read
- Prefer a local class if
  - ⇒ More than one instance of the class is required