What is Java?

- ".. A **simple**, **pure object-oriented**, **distributed**, **interpreted**, **robust**, **secure**, **architecture neutral**, **portable**, **high-performance**, **multi-threaded**, and **dynamic** language."

  [Sun Microsystems, Summer 1995]

Two Application Styles

- **Application**:

  Program that executes using the java **interpreter**.

- **Applet**:

  1. Program that runs in –
     a. **appletviewer** (test utility for applets).
     b. Web browser (IE, NSCP Communicator).
  2. Executes when HTML (HyperText Markup Language) document containing applet is opened and downloaded.

Applications vs Applets

- Applications are more robust:
  1. Applications do not have the same security restrictions that applets have.
  2. Applications can be compiled with JIT (Just-In-Time) and then execute as fast as C++.
  3. Applications can do RMI, JDBC, JavaBeans.

Applications vs Applets (cont)

- Applets give life to Web pages:
  1. Applets have a large library of AWT classes available.
  2. Applets provide the function of an application without using disk space.
  3. Applets do not have a "**main**" method.

Running Java Programs

- **Java Virtual Machine (JVM)**
  - Program running on computer.
  - Simulates a virtual computer running Java.
- **JVM loads classes dynamically**
  - Upon first reference to class in program.
  - Looks at directory / jar files in CLASSPATH.
- **CLASSPATH**
  - Colon separated list of names.
  - Example:

    CLASSPATH = . : $HOME/Java

Jar Files

- Zip file containing 1 or more **.class** files.
- Useful for bundling many Java files.
- Treated by JVM as an entire directory.

- Create using:

  jar cf [filename] [files / directories to put in jar]

Java vs C++

- Java = C++
  - { struct, union, pointer, operator overloading, multiple inheritance, … }
  + { strict type checking, automatic garbage collection, … }

Procedural Programming in Java

- ***public class HelloWorld {***

  public static void main(String [] args) {
  System.out.println("Welcome to Java" + args[0]);
  } // end method *main*

} // end class *HelloWorld*

- All programs are placed inside a 'class'.
- Class names start with an upper-case letter.
- Java is case sensitive (capitalization matters).

Example Program (2)

- The code is placed in a file called *HelloWorld.java*.
- The program is always called ***main***.
- The class is delimited by braces { … }
- Java is free format – spaces, tabs & blank lines ignored.
- The program – ***main*** – is enclosed within braces { … }
- Character strings enclosed between quote marks “…”.

Example Program (3)

- *System.out* refers to *stdout*.
- *println* is a method (procedure) to output to the screen.
- Text following // is comment & ignored by compiler.
- As a parameter to ***main***, *args* holds command-line arguments.
- *args[0]* refers to the first element of the array of strings.
- + represents string concatenation.

Compile and Execute

- Creating source file – *HelloWorld.java*
  Use vi, emacs, or other text-editor

- Compiling program – to create bytecode file *HelloWorld.class*
  javac HelloWorld.java

- Running program
  java HelloWorld

Static Variable Initialization

- **Initialization Order**
  1. Static initializations
  2. Initialization block
  3. Constructor
- **Example**

```
class Foo {
    static Integer A = new Integer (1);        // static init
    { A = new Integer (2); }                   // init block
    public Foo( ) { A = new Integer (3); }     // constructor
}
```

Static Variable Initialization Example

```
class Foo {
    static Integer A = new Integer (1);        // 1) static init
    { A = new Integer (2); }                   // 2) init block
    public Foo( ) { A = new Integer (3); }     // 3) constructor

    public static void main (String args[]) {
        System.out.print("A=“ + A + "\n"); // prints A=1
        Foo f = new Foo();
        System.out.print("A=“ + A + "\n"); // prints A=3
     }
```

Strings in Java

- Creating string variables:

  String greeting = “Hello”;

  String text; text = args[0];

- Comparing strings:

  text.equals(“Hello”)

  text.compareTo(args[1])

- Determining the length of strings:

  int len = text.length();

- Concatenating strings:

  String message = text + ‘-’ + args[1];

Strings in Java (cont)

- Picking off parts of a string:

  char ch = text.charAt(0);

  message = text.substring(3, 6);

- Changing case:

  String big = text.toUpperCase();

- The **String** class has 48 methods. Take a careful look at these by reference to

  java.lang -> String

Basic Data Types in Java

- Integer (whole number) types:

  byte tiny; // 8 bits, range -128 .. +127

  short x, y = 5, z; // 16 bits, range -32768 .. +32767

  int count = 0;

  // 32 bits, range approx +-2000,000,000

  long dist = -53; // 64 bits, range approx +-10 zillion

- Fractional values (floating-point) types:

  float height = -3.4e22; // 32 bits, 7 significant digits

  double fine = 123.5; // 64 bits, 15 significant digits

Basic Data Types (cont)

- Single characters:
  char ch = 'X'; // 16 bits - Unicode character
- Logical (truth) values:
  boolean ok = true; // 1 bit - constants - true, false

- Conversion (casting) between types:
  Automatic if no loss of info,
  e.g. ***count = tiny; fine = dist;***
  Explicit otherwise,
  e.g. ***tiny = (byte)count; dist = (long)fine;***

Command-Line Arguments

- Determining number of arguments:

  int numberOfArgs = args.length;

  // Note - no brackets

  For any array, *arrayName.length* gives length.
- Conversion to integer:

  int value = Integer.parseInt(args[0]);

  parseInt() raises *NumberFormatException* if a non-numeric value is entered.
- Conversion to float:

  double volume = Double.parseDouble(args[1]);

  Again this can raise *NumberFormatException*.

Command-Line Arguments (cont)

- Dealing with exceptions:

```
try {
    value = Integer.parseInt(args[0]);
} // end try
catch(NumberFormatException nfe) {
   System.err.println("Argument should be an
                                integer");
   System.exit(-1);
} //end catch
```

Boolean Variables

- Boolean variables contain truth values and can be used directly:

```
boolean  ok = false;
text.length() < 10 && count < 25 && ok
        // always evaluates to false
ok = count != 100 && ch != 'X';
        // assigning a value
```

Sub-Programs in Java

- Called – subroutines (FORTRAN), procedures (PASCAL), functions (C) or methods (Java), but all refer to sub-programs.
- Subdivide a large program into smaller pieces – easier to understand.
- Allow several people to work on different parts of the program at once.
- Factor out commonly occurring pieces of code and write only once.

Sub-Programs (cont)

- Avoid repetition and hence changes only made in one place.
- Create sub-programs which can be re-used in other applications.
- Minimize the amount of re-compilation when changes are made.

Stream Input / Output

- Bytes: InputStream, OutputStream.
- Character: FileReader, PrintWriter.

Usage:

- ***import java.io.*;***
- **Open stream connection.**
- **Use stream -> read or/and write; catch exception if needed.**
- **Close stream.**

Standard Input / Output

- Standard I/O
  - Provided in **System** class in **java.lang**
  - **System.in**
    - An instance of InputStream
  - **System.out**
    - An instance of PrintStream
  - **System.err**
    - An instance of PrintStream

Prompted Keyboard Input

```
import java.io.*;          // provides visibility of I/O facilities
public class  Greeting {
    public static void main(String [] args) {
        String name;
        try {
            InputStreamReader isr =
                new InputStreamReader(System.in);
            BufferedReader in = new BufferedReader(isr);
            System.out.print("Enter your name : ");
                // prompt for input
            name = in.readLine();          // get it
        } // end try
        catch(IOException ioe){}
        System.out.println("Welcome to Java - " + name);
    } // end main
} // end class Greeting
```

Reading Text Files

```
import java.io.*;
public class FileReadDemo {

    public static void main(String [] args)
                            throws IOException {
        FileReader fr = null;
        try {
                fr = new FileReader(args[0]);
        } // end try
        catch(FileNotFoundException fnf) {
                System.err.println("File does not exist");
                System.exit(-1);
        } // end catch
```

Reading Text Files (cont)

```
        BufferedReader inFile = new BufferedReader(fr);
        String line;
        while (inFile.ready()) {
                line = inFile.readLine();
                System.out.println(line);
        } // end while
        inFile.close();
    } // end main

} // end class FileReadDemo
```

Writing Text Files

```
import java.io.*;
public class FileWriteDemo {

    public static void main(String [] args) throws IOException {
        FileOutputStream fos = new FileOutputStream(args[0]);
        PrintWriter pr = new PrintWriter(fos);
        for ( int i = 0; i < 20; i++)
            pr.println("Demonstration of writing a text file");
        pr.flush();
        pr.close();
    } // end main

} // end class FileWriteDemo
```

Debugging

- **Process of finding and fixing software errors**
  - After testing detects error.
- **Goal**
  - Determine cause of run-time & logic errors.
  - Correct errors (without introducing new errors).
- **Similar to detective work**
  - Carefully inspect information in program
    - Code
    - Values of variables
    - Program behavior

Debugging – Approaches

- **Classic**
  - Insert debugging statements
  - Trace program control flow
  - Display value of variables

- **Modern**
  - IDE (integrated development environment)
  - Interactive debugger

Interactive Debugger

- Capabilities
  - Provides trace of program execution.
  - Shows location in code where error encountered.
  - Interactive program execution
    - Single step through code.
    - Run to breakpoints.
  - Displays values of variables
    - For current state of program.

Interactive Debugger (2)

- Single step
  - Execute single line of code at a time.
  - When executing **method**, can
    - Finish entire method.
    - Execute first line in method.
  - Tedious (or impractical) for long-running programs.

Interactive Debugger (3)

- Breakpoint
  - Specify location(s) in code.
  - Execute program until breakpoint encountered.
  - Can skip past uninteresting code.