INTRODUCTION TO JAVA PL

By Pakita Shamoi

Java

- Java is a general-purpose object-oriented language. It is intended to let application developers "write once, run anywhere"

- Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

Java features

- **Simple (the language itself)**
- **Object oriented**
  - focus on the data (objects) and methods manipulating the data
  - all functions are associated with objects
  - almost all datatypes are objects (files, strings, etc.)
- **Portable**
  - same application runs on all platforms
- **Dynamic Memory Management**
  - garbage collection

Java features

- **Reliable**
  - extensive compile-time and runtime error checking (e.g. bytecode cheking)
  - no pointers but real arrays. Memory corruptions or unauthorized memory accesses are impossible
  - automatic garbage collection tracks objects usage over time
- **Interpreted**
  - java compiler generate byte-codes, not native machine code
  - the compiled byte-codes are platform-independent
  - java bytecodes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)
- **Network support**

Java features

- **Secure**
  - usage in networked environments requires more security
  - memory allocation model is a major defense
  - access restrictions are forced (private, public)
  - array access bounds checking
- **Multithreaded**
  - multiple concurrent threads of executions can run simultaneously
- **Dynamic**
  - java is designed to adapt to evolving environment
  - libraries can freely add new methods and instance variables without any effect on their clients
  - interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented

Portability

- Portability means that programs must run similarly on any supported hardware/OS platform. This is achieved by compiling the Java code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code.
- Bytecode instructions are like machine code, but are intended to be interpreted by a JVM written specifically for the host hardware.
- A major benefit of using bytecode is porting. However, interpreted programs almost always run more slowly than programs compiled to native executables.
- Just-in-Time (JIT) compilers compile bytecodes to machine code during runtime.

JVM

- The JVM is a piece of software written specifically for a particular platform.

- At run time, the JVM reads and interprets .class files and executes the program's instructions on the native hardware platform for which the JVM was written.

- The JVM is the heart of the Java language's "write-once, run-anywhere" principle. Your code can run on any chipset for which a suitable JVM implementation is available.

- JVMs are available for major platforms like Linux and Windows

Java Runtime Environment - JRE

- The Java Runtime Environment includes the JVM, code libraries, and components that are necessary for running programs written in the Java language.

- It is available for multiple platforms

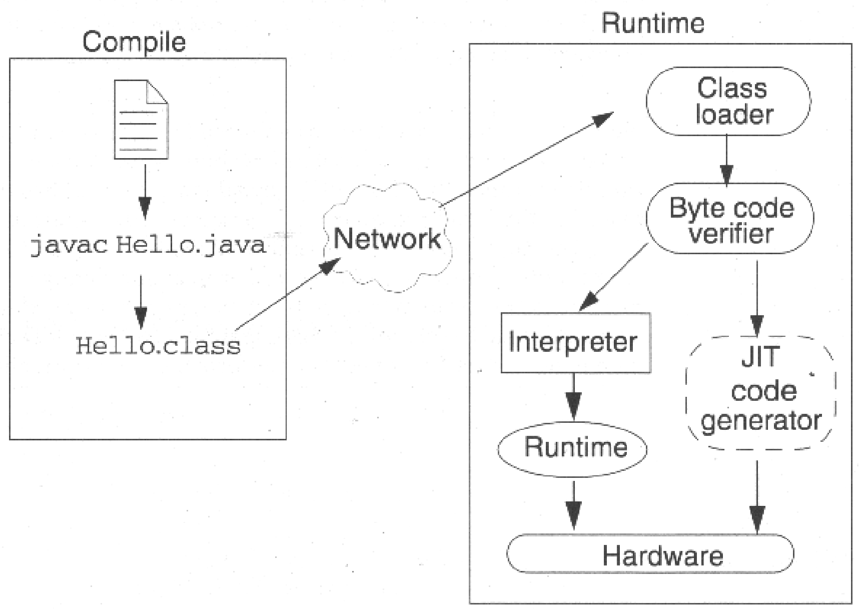
Performance issues

- Using a VM is slower than compiling to native instructions.
  - JIT compilers convert Java Bytecode to machine language during runtime.

- Safety/Security slow things down
  - all array accesses require bounds check
  - Many I/O operations require security checks

Compilation, launching

- **`javac`**: the Java compiler.
  - Reads source code and generates bytecode.
- **`java`**: the Java interpreter
  - Runs bytecode.



Java compiler

- We write source code in .java files and then compile them. The compiler checks your code against the language's syntax rules, then writes out bytecodes in .class files.
- Bytecodes are standard instructions targeted to run on a Java virtual machine (JVM). In adding this level of abstraction, the Java compiler differs from other language compilers, which write out instructions suitable for the CPU chipset the program will run on.
- Usage: **`javac`** ***`filename`*** **`.java`**
  - You can also do: **`javac *.java`**
  - Creates ***`filename`*** **`.class`** (if things work)

Java interpreter

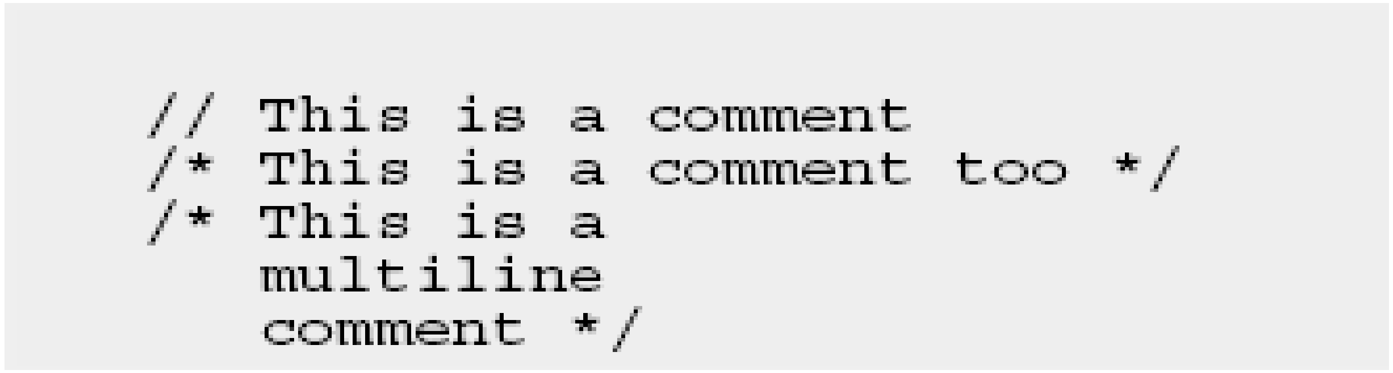
- Usage: **`java`** ***`classname`***

You tell the interpreter a class to run, not a file to run!

- The named class should have a method with prototype like:
  - **`public static void main()`**

Comments

- A single line style marked with two slashes (//)
- A multiple line style opened with /* and closed with */
- The commenting style opened with /** and closed with */



```
// This is a comment
/* This is a comment too */
/* This is a
   multiline
   comment */
```

First java program - text-printing

```
public class Welcome1
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      System.out.println( "Welcome to Java Programming!" );

   } // end method main

} // end class Welcome1
```

```
Welcome to Java Programming!
```

- When you save your public class declaration in a file, the file name must be the class name followed by the "**.java**" file-name extension. For our application, the file name is Welcome1.java.
- **System.out** is known as the **standard output object**

First java program - text-printing

- To compile the program, type

```
javac Welcome1.java
```

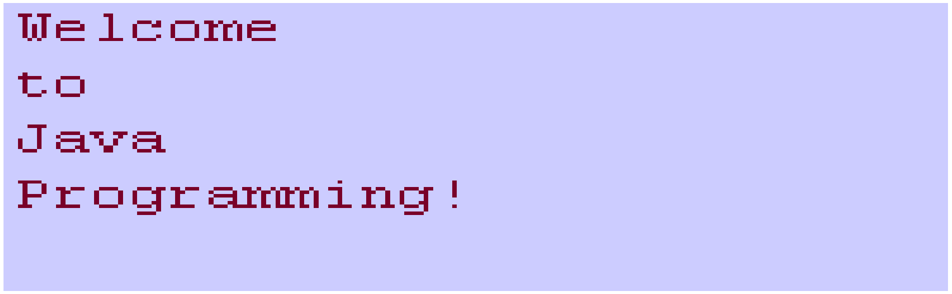
- If the program contains no syntax errors, the preceding command creates a new file called Welcome1.class (known as the **class file** for Welcome1) containing the Java bytecodes that represent our application.

- When we use the `java` command to execute the application, these bytecodes will be executed by the JVM.

```
java Welcome1
```

- To print on separate lines, use

```
System.out.println( "Welcome\nto\nJava\nProgramming!" );
```



```
Welcome
to
Java
Programming!
```

- `System.out.print()` method – just prints passed text without moving output cursor to a new line

Addition program

```java
// Fig. 2.7: Addition.java
// Addition program that displays the sum of two numbers.
import java.util.Scanner; // program uses class Scanner

public class Addition
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int number1; // first number to add
      int number2; // second number to add
      int sum; // sum of number1 and number2

      System.out.print( "Enter first integer: " ); // prompt
      number1 = input.nextInt(); // read first number from user

      System.out.print( "Enter second integer: " ); // prompt
      number2 = input.nextInt(); // read second number from user

      sum = number1 + number2; // add numbers

      System.out.printf( "Sum is %d\n", sum ); // display sum

   } // end method main

} // end class Addition
```

Import declaration

- Helps the compiler to locate a class used in program.

- A great strength of Java is its rich set of predefined classes that programmers can reuse rather than "reinventing the wheel." These classes are grouped into **packages** named collections of classes. Collectively, Java's packages are referred to as the **Java Application Programming Interface** (Java API).

- In our example we use Java's predefined **Scanner** class (discussed shortly) from package **java.util.**

Variables

- A **variable** is a location in the computer's memory where a value can be stored for use later in a program.
- All variables must be declared with a name and a type before they can be used.

- **Variable declaration statement** specifies the name and type of a variable used in program.

```
int count;
```

- **Variable assignment statement**

```
count = 8;
```

Primitive & reference types

- **The primitive types** are the boolean type and the numeric types. The numeric types are the integral types - *byte, short, int, long, and char*, and the floating-point types - *float and double*.

- **The reference types** are *class* types, *interface* types, and *array* types. There is also a special null type.

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

Primitive & reference types

- **Primitive values .** The value of a variable of primitive type can be changed only by assignment operations on that variable. ALWAYS use “==” (double equal) operator to compare for equality.

- An object is a class instance or an array. The **reference values** are pointers to these objects, and a special null reference, which refers to no object.

- Values of primitive types are either stored directly in fields (for objects) or on the stack rather than on the heap, as commonly true for objects. This was a conscious decision by Java's designers for performance reasons

Java API

- A Web-based version of this documentation can be found at

java.sun.com/j2se/5.0/docs/api/

- Also, you can download this documentation to your own computer from

java.sun.com/j2se/5.0/download.jsp

The download is approximately 40 megabytes (MB) in size

Home Work

- JHTP – Chapter 2 (2.1-2.3, 2.5-2.8)