

Features of Java:

Java is completely object oriented. C++, being more compatible to C, allows code to

exist outside classes too. However in Java, every line of code has to belong to

some or other class. Thus it is closer to true object oriented language.

Java is simpler than C++, since concepts of pointers or multiple inheritance do not

exist.

Let us discuss these features in detail.

Object-Oriented:

To stay abreast of modern software development practices, Java is Object-Oriented

from the ground up. Many of Java’s Object-Oriented concepts are inherited from

C++, the language on which it is based, plus concepts from other Object-Oriented

languages as well.

Simple:

Java omits many confusing, rarely used features of C++. There is no pointer level

programming or pointer arithmetic. Memory management is automatic. There are

no header files, structures, unions, operator overloading, virtual base classes, and

multiple inheritance.

Robust:

Java programs are reliable. Java puts a lot of emphasis on early checking for

potential problems, dynamic checking and eliminating situations that are errorprone.

Java has a pointer model that eliminates possibility of overwriting memory

and corrupting data.

Security:

Java is intended to be used in networked / distributed environments. Thus a lot of

emphasis is placed on security. Normally two things affect security:

• confidential information may be compromised, and

• computer systems are vulnerable to corruption or destruction by hackers

Java’s security model has three primary components:

• Byte code Verifier: The byte code verifier ensures the following:

 the Java programs have been compiled correctly,

 they will obey the virtual machine’s access restrictions, and

 the byte codes will not access private data when they should not

• Class Loader: When the loader retrieves classes from the network, it

keeps classes from different servers separate from each other and from

local classes. Through this separation, the class loader prevents a class

that is loaded off the network from pretending to be one of the standard

built-in classes, or from interfering with the operation of classes loaded from

other servers.

• Security Manager: It implements a security policy for the VM. The security

policy determines which activities of the VM is allowed to perform and under

what circumstances, operation should pass.

Architecture-Neutral:

A central issue for the Java designers was of code longevity and portability. One of

the main problems facing programmers is that there is no guarantee that when you

write a program today, it will run tomorrow — even on the same machine.

Operating system upgrades, processor upgrades, and changes in core system

resources can all combine to make a program malfunction. The Java designers

made several hard decisions in the Java language and the Java Virtual Machine

(JVM) in an attempt to alter this situation. Their goal was “write once; run

anywhere, any time, forever”.

To a great extent, this goal was accomplished.

Platform-Independent:

This refers to a program’s capability of moving easily from one computer system to

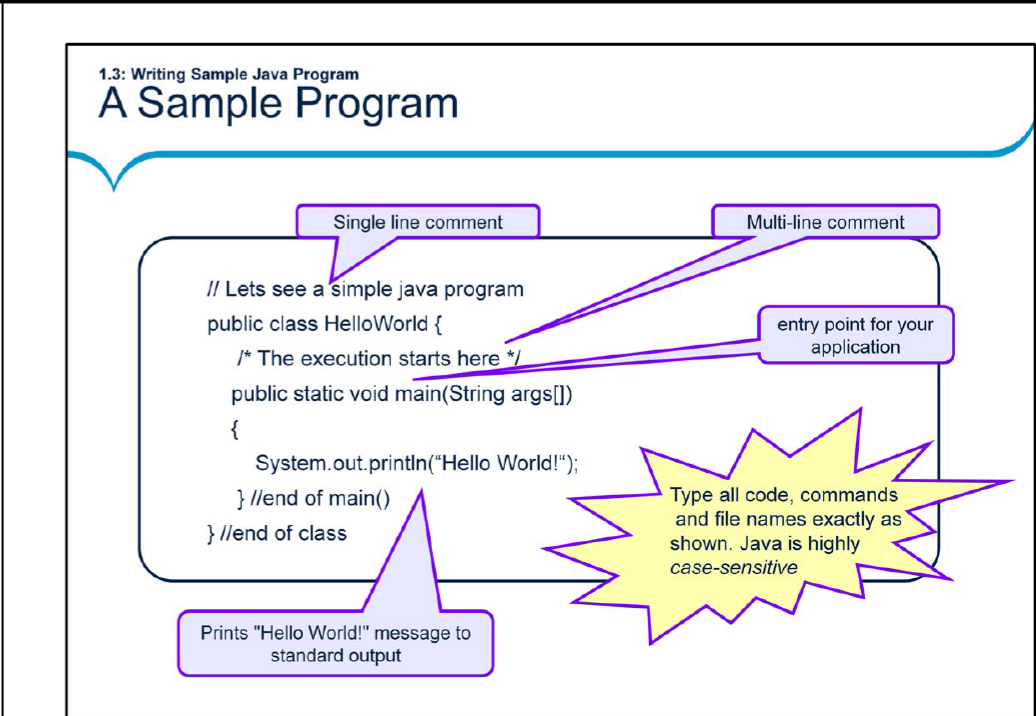
another. Java is platform-independent at both the source and the binary level.

 At the source level, Java’s primitive data types have consistent sizes across

all development platforms.

 Java binary files are also platform-independent and can run on multiple

platforms without the need to recompile the source because they are in the

form of byte codes.

Introduction to Java Features – A Sample Program:

• Here is our first Java program..

• Let us have a closer Look at the “Hello World!” program:

 class HelloWorld begins the class definition block for the “Hello

World!” program. Every Java program must be contained within a

“class” block.

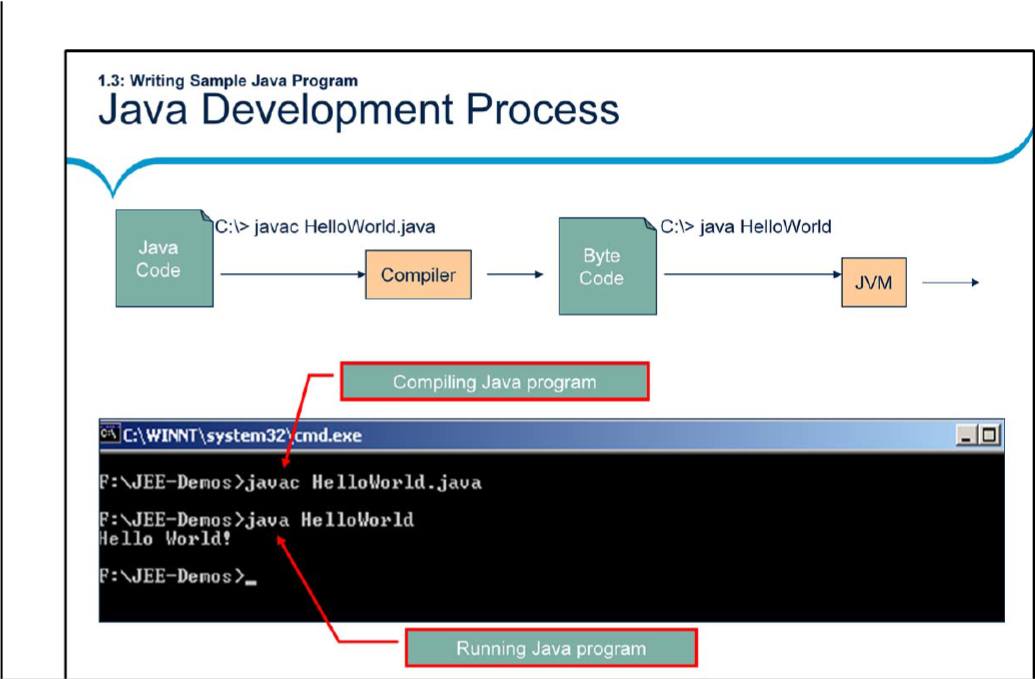
 public static void main(String[] args) { …} is the entry point for your

application. Subsequently, it invokes all the other methods required by

the program.

• This program when executed will display “Hello World” on the screen. We shall

see this in the next slide.



Introduction to Java Features – Java Development Process:

The Java Development Process involves the following steps:

1. Write the Java code in a text file with a .java extension, namely

“HelloWorld.java”. By convention, source file names are named by the public

class name they contain.

2. At the command line, compile the code using the Java compiler (javac). The

javac compiler converts the source into Byte Codes and stores it in a file having

.class extension. What is special about byte codes? Unlike traditional compilers,

javac does not produce processor specific native code. Instead it generates

code which is in the language of JVM (Java Virtual Machine).

Note: To have access to the javac and the java commands, you must set your

path first. To do so, you may type the following at the command prompt:

Set path=<your java-home directory>\bin

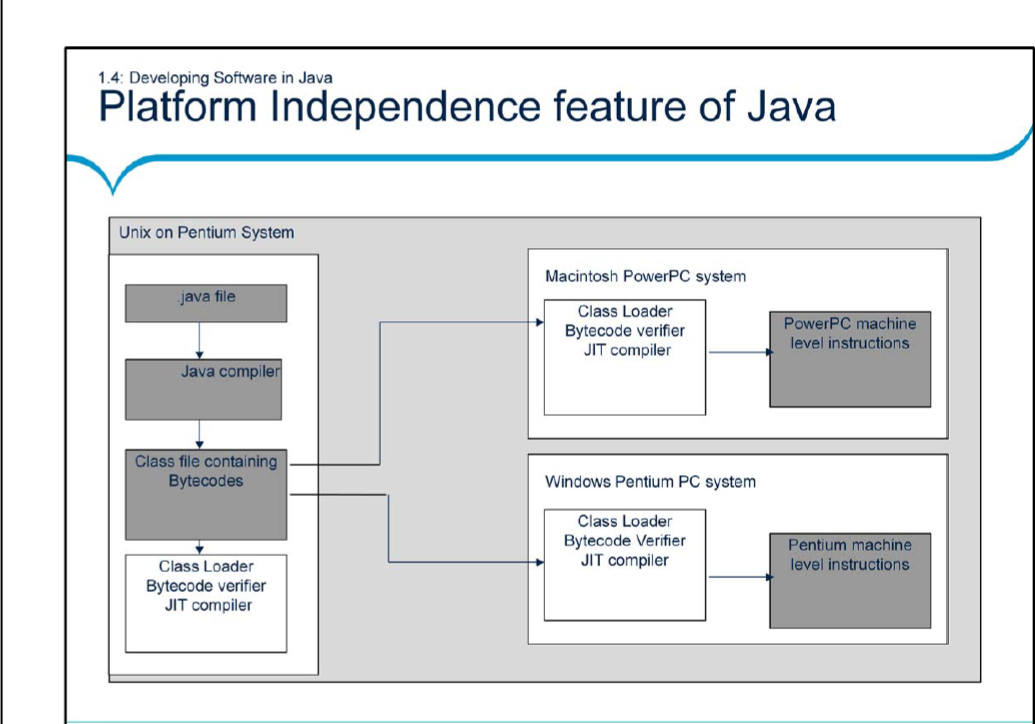
For example: Set path= C:\Program Files\Java\jdk1.8.0\_25 \bin

We can also set the environment variables (path and classpath) through the

Control Panel.

3. To run this program, use the Java command with class file name as the

command parameter as shown on the above slide. The output of the program

would, of course, be “Hello World!” Platform Independence:

•The figure illustrates how platform independence is achieved using Java. Once you

write Java code on a platform and run it through Java Compiler, the class file

containing byte codes is obtained.

•Different JVMs are available for different platforms. So the JVM for Unix on

Pentium will be different from the JVM for Mac or for Windows. Each of these JVMs

take the same input, namely the Class File, and produce the machine level

instructions for the respective platforms.

•One common grouse among developers is that Java programs take longer to

execute because the compiled bytecodes are interpreted by the JVM. The Java justin-

time (JIT) compiler, compiles the bytecode into platform-specific executable code

(native code) that is immediately executed, thus speeding up execution! Traditional

native code compilers run on the developer’s machine and are used by

programmers, and produce non-portable executables. JIT compilers run on the

user’s machine and are transparent to the user. The resulting native code

instructions do not need to be ported because they are already at their destination.

JVM:

When you compile a Java program (which usually is a simple text file with .java

extension), it is compiled to be executed under VM. This is in contrast to

C/C++ programs, which are compiled to be run on a real hardware platform,

such as a Pentium processor running on, say Win 95. The VM itself has

characteristics very much like a physical microprocessor. However, it is

entirely a software construct. You can think of the VM as an intermediary

between Java programs and the underlying hardware platform on which all

programs must eventually execute.

• Even with the VM, at some point, all Java programs must be resolved to a

particular underlying hardware platform. In Java, this resolution occurs within

each particular VM implementation. The way this works is that Java programs

make calls to the VM, which in turn routes them to appropriate native calls on

the underlying platform. It is obvious that the VM itself is very much platform

dependent.

How does the JIT compiler work?

• The VM instead of calling the underlying native operating system, it calls the

JIT compiler. The JIT compiler in turn generates native code that can be

passed on to the native operating system for execution. The primary benefit

of this arrangement is that the JIT compiler is completely transparent to

everything except VM. The neat thing is that a JIT compiler can be integrated

into a system without any other part of the Java runtime system being

affected.

• The integration of JIT compilers at the VM level makes JIT compilers a

legitimate example of component software. You can simply plug in a JIT

compiler and reap the benefits with no other work or side effects.

• A Java enabled browser contains its own VM. Web documents that have

embedded Java applets must specify the location of the main applet class file.

The Web browser then starts up the VM and passes the location of the applet

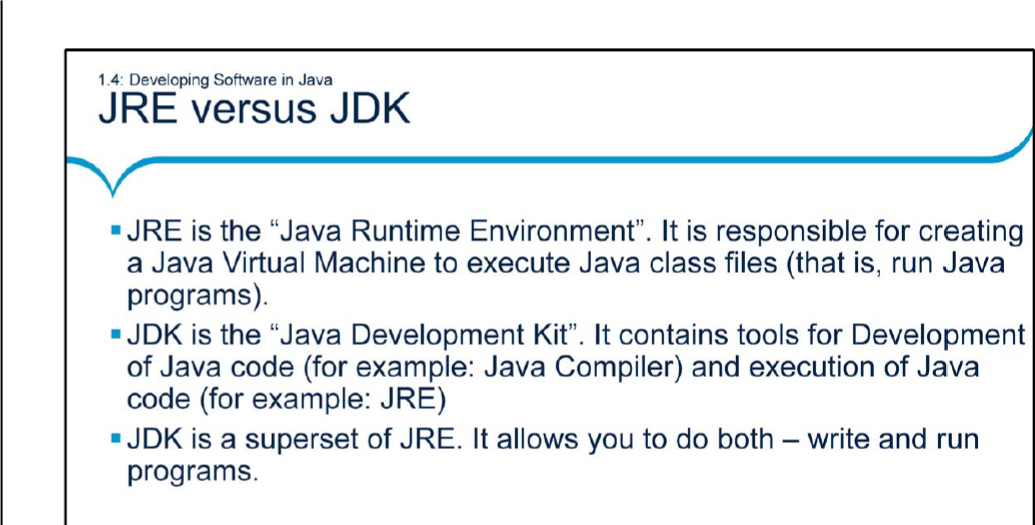
class file to the class loader. Each class file knows the names of any

additional class files that it requires. These additional class files may come

from the network or from client machine. Supplement classes are fetched

only if they are actually going to be used or if they are necessary for the

verification process of the applets.



Difference between JRE and JDK:

• The Java Development Kit (JDK) is a superset which includes Java

Compilers, Java Runtime Environments (JRE), Development Libraries,

Debuggers, Deployment tools, and so on. One needs JDK to develop Java

applications. We have different versions that include JDK 1.2, JDK 1.4, and so

on.

• The Java Runtime Environment (JRE) is an implementation of JVM that

actually executes the Java program. It is a subset of JDK. One needs JRE to

execute Java applications.

