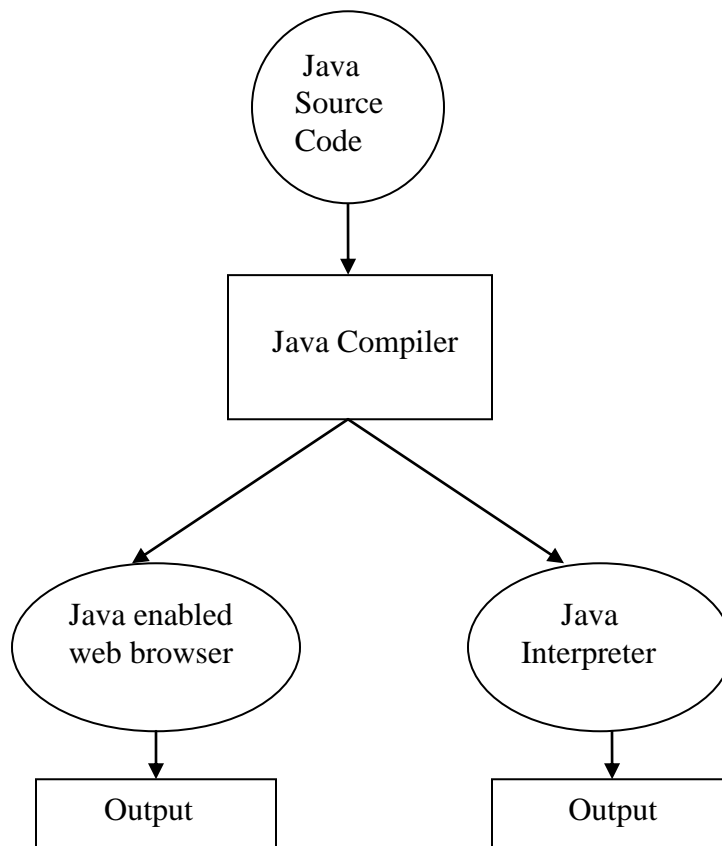
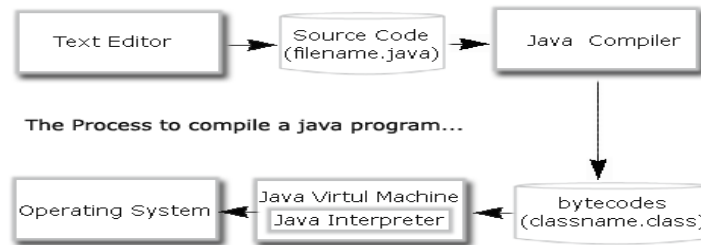


Introduction to Java

- **The Creation of Java-** Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. This language was initially called “Oak” but was renamed “Java” in 1995.
- **Java Applets and Applications:** Java can be used to create two types of programs: applications and applets.
 - ✓ An **application** is a program that runs on your computer, under the operating system of that computer. That is, an application created by Java is more or less like one created using C or C++. When used to create applications, Java is not much different from any other computer language.
 - ✓ An **applet** is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. The important difference is that an applet is an *intelligent program*, not just an animation or media file. In other words, an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over.



- **Java Compilation-**



- ✓ **Byte code & JVM-** The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is byte code. *Byte code* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*. That is, in its standard form, the JVM is an *interpreter for byte code*. This may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted—mostly because of

performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why.

Translating a Java program into byte code helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java byte code.

If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of byte code is the easiest way to create truly portable programs.

The fact that a Java program is interpreted also helps to make it secure. Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. The use of byte code enables the Java run-time system to execute programs much faster than you might expect.

Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of byte code into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for byte code, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles byte code into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire Java program into executable code all at once,

because Java performs various run-time checks that can be done only at run time. Instead, the JIT compiles code as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to byte code, the portability and safety features still apply, because the run-time system (which performs the compilation) still is in charge of the execution environment. Whether your Java program is actually interpreted in the traditional way or compiled on-the-fly, its functionality is the same.

- **The Java Buzzwords**

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Security: As you are likely aware, every time that you download a “normal” program, you are risking a viral infection. Prior to Java, most users did not download executable programs frequently, and those who did scan them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. Java answers both of these concerns by providing a “firewall” between a networked application and your computer.

When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most important aspect of Java.

Portability: Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically

downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. Now, the same mechanism that helps ensure security also helps create portability. Indeed, Java's solution to these two problems is both elegant and efficient.

Simple: Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left out of Java or implemented in a cleaner, more approachable manner.

Object-Oriented: Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non objects.

Robust: The multi platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java.

Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented

exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded: Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral: A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if 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 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.

Interpreted and High Performance: As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs. As explained earlier, while it is true that Java was engineered for interpretation, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. “High-performance cross-platform” is no longer an oxymoron.

Distributed: Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra address space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called *Remote Method Invocation (RMI)*. This feature brings an unparalleled level of abstraction to client/server programming.

Dynamic: Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the

robustness of the applet environment, in which small fragments of byte code may be dynamically updated on a running system.