

# *Java as a Programming Language of Choice*

---

Why should you learn Java? There are many answers to this question. In a relatively short amount of time, Java has established itself as an extremely powerful and versatile programming language that incorporates a lot of syntax, semantics and philosophies of other successful programming languages.

This chapter attempts to answer the question of why one should learn Java by comparing Java to some of its predecessors, as well as describe how Java promotes good programming practices.

---

## *Java Enforces Good Programming Practices*

In later chapters, we will see that Java has rigorous compilation and run-time rules that help to enforce good programming practices and habits. Such practices are generally not enforced in other languages, which implies that it is up to the programmer to choose to apply such practices.

As an example, consider the C++ language (or its predecessor, C). Although it is fairly easy to successfully compile a C or C++ program, debugging the program or application and getting it to actually work as expected tends to be a relatively large undertaking (i.e. relative to a comparable Java program).

Compilation rules in C and C++ are not very rigorous\* and the implicit assumption is that the programmer “knows what he or she is doing.” Even with a programmer’s best effort, this may not always be a very good assumption and can result in hidden (or blatant) logic errors, including the infamous memory leak problem (a problem that does not occur in Java).

As another example, consider the Perl language. Although Perl is a very powerful language with exceptional pattern matching abilities, its compilation rules, when compared to Java, tend to be extremely lenient. Java (as well as C and C++) are *strongly typed* languages, meaning that variables and their corresponding types (e.g. integer, character, etc.) must be explicitly declared by the programmer before being used in a program. Perl is not a strongly typed language, which in many ways is an unnecessary and often misleading level of flexibility.

---

### *Java Promotes Object-Oriented Programming*

Beyond the immediate gain of enforcing good programming practices, learning and using Java also forces you to think in a more *object-oriented* way, which generally lends itself to representing real-world problems more easily and effectively. Many prominent languages today are considered *functional* or *modular* languages, including C, COBOL, Pascal, Fortran, etc. Such languages provide constructs to break down a problem into its underlying *functions*. Each function has a set of inputs, a set of outputs, and an algorithm used to convert the inputs into the outputs. The algorithm may also produce *side-effects*, including such tasks as displaying information to the user, manipulating data within a database, etc.

Such functions are usually organized into a multitude of *modules*. Each of the modules is typically designed with a specific theme in mind. For example, an input/output module may contain functions to open and close files, print data to the screen, receive input from the keyboard, etc. A math module may consist of many mathematical functions, including the square root function, a function to round numbers to specific decimal places, trigonometric functions, etc.

---

\* Note that compiler options may be used to force more rigorous compilation rules to be in effect; however, it is still up to the programmer’s discretion as to whether to heed or ignore warnings produced in these modes.

In general, such functional languages coerce the programmer (and designer) to think in terms of functions, which in many cases is not the “natural” way to organize a solution to a problem. The object-oriented approach directs the programmer to think in terms of *objects*, which essentially represent the data of a system and the corresponding operations on the data.

---

### *Java Claims Platform Independence*

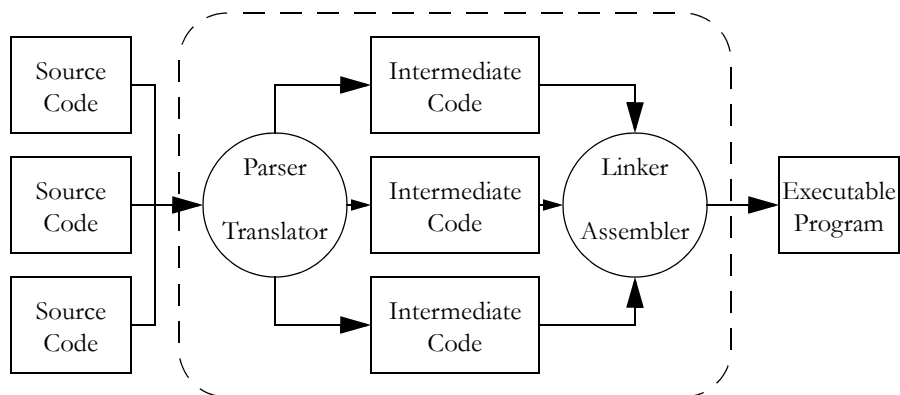
As described in more detail below, Java achieves *platform independence* by making use of a Java Virtual Machine (JVM). By platform independence, we mean that Java code may be developed on any platform (e.g. Unix, Windows, etc.) and will subsequently execute in a consistent manner on any platform.

### Compiled Languages and Compilers

In many modern programming languages, source code is translated into executable machine code by making use of a *compiler*. Common programming languages that are compiled into platform-specific executable machine code include Pascal, Fortran, C and C++. Such languages are often referred to as *compiled languages*.

As Figure 1-1 illustrates, a compiler is a software component that parses source code for a specific programming language, transforms the source code into a simplified intermediate form, and then generates an executable machine language program for a specific hardware platform.

FIGURE 1-1. A simple compiler.



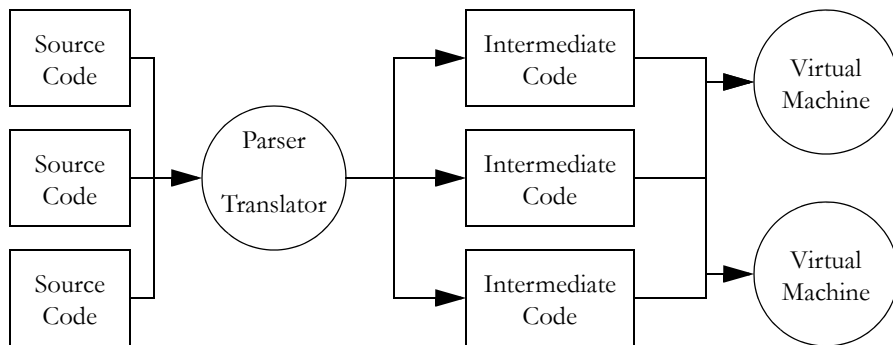
More specifically, source code files are parsed and translated into intermediate code, which usually stays hidden within the compiler software itself. Once constructed, the intermediate code is linked and assembled into an executable program that is built specifically for a single hardware platform. Because machine languages differ based on the underlying microprocessor of the corresponding machine, an executable program on one platform will usually not execute on a different platform, or even a different generation or version of the same platform.

Most good compilers perform numerous optimizations to the intermediate code and subsequently generated machine language code, taking full advantage of specific features of the given machine language. Given these optimizations, the resulting program generally executes very quickly on the given hardware platform. Moving to a different hardware platform is essentially impossible without first recompiling the source code on the new hardware platform. For most programming languages, porting source code from one platform to another is a long and error-prone endeavor. Application developers and software engineers spend countless hours struggling with porting issues due to language and operating system differences across heterogeneous platforms.

### Interpreted Languages and Virtual Machines

While many programming languages make use of a compiler, numerous programming languages are executed via an *interpreter*, which is a software component that executes a program a single instruction at a time by interpreting what is usually an intermediate form of the initial source code. Figure 1-2 shows how an *interpreted language* works.

**FIGURE 1-2. An interpreted language generally separates the compilation process from the execution process.**



An abbreviated compilation process is still usually necessary in an interpreted language; however, this compilation process does not proceed beyond the generation of intermediate code. Executing this intermediate code requires the use of an interpreter, or *virtual machine*. The virtual machine interprets the intermediate code one instruction at a time, executing one instruction before interpreting and executing the next instruction.

Executing a program via an interpreter is generally more time-consuming than that of a fully-compiled executable program, which may be a disadvantage for time-sensitive applications. The primary advantage to providing virtual machines on a wide variety of platforms is that the same intermediate code may execute on a heterogeneous set of machines with theoretically no changes to the initial source code. As an example, Java intermediate code (better known as *byte-code*) may be made available on a website and subsequently executed on a wide variety of network browsers that provide a *Java Virtual Machine (Java VM or JVM)*.

In summary, the portability issues that are present in fully-compiled languages are substantially reduced when using interpreted languages, especially Java. Common interpreted languages include Lisp, Perl and of course Java. Note that many full-scale compilers exist for interpreted languages, allowing the programmer to take advantage of the performance gains of the compiled executable, while potentially sacrificing platform independence. Such compilers are often called *native compilers*, since they produce an executable program that is a “native” to a specific hardware platform.

In addition, later versions of the Java VM have been optimized to provide better performance. Specific Java VMs also make use of *just-in-time (JIT)* compilers, which translate the intermediate byte-codes into executable native machine language while a Java program is running.

---

### *Java Standardizes Graphical Components*

From its inception, the Java programming language has included a strong and diverse library, which encourages programmers to use existing software components instead of spending the time to create their own. In other words, why “recreate the wheel” when a consistent and proven solution is already available within the Java programming environment?

Lacking in other programming environments, the Java programming environment also includes a standardized graphics library that provides platform-independent graphical

constructs. In the mid to late 1990s, this opened the door for graphical Java programs called *applets* to execute on a multitude of network browsers and hardware platforms.

### **Applets vs. Applications**

As an interpreted language, Java source code is compiled into intermediate byte-code, which is subsequently executed via the Java VM. In the mid to late 1990s, Java VMs were integrated into leading web browsers at Java's inception, allowing a great number of Java applets to be downloaded and executed on numerous client machines. Java applets provide a programmer with far more flexibility than that of the standard non-dynamic HTML.

However, downloading a program from a website on the Internet and executing that program on your own machine can be dangerous, as it may accidentally or maliciously damage your computer (e.g. by introducing a virus into your system). With such security risks being a primary deterrent to using the Internet, Java applets were designed to be fully secure and trustworthy. The Java VM does not allow an untrustworthy program to be executed. Further, Java applets are not allowed to access local files or execute local programs on the client machines. With the absence of direct pointer access and the inability to traverse past the boundaries of an array, Java programs are fully contained within their own confined memory space on a client machine.

Although most of the initial excitement surrounding Java has been focused on its ability to reside within various web pages, Java is also used as a full-fledged programming language to develop full-scale applications. A Java application does not suffer the same security-related restrictions as that of a Java applet, though a Java application does not reside within a web page.

---

### *Chapter Summary*

Throughout this introductory chapter, we have seen that Java has been designed as a comfortable language for programmers with a

### **Cheat Sheet**

---

## Chapter Summary

---

### **Bibliographic Notes**

For a more in-depth look at compilation and interpretation, refer to [SCOTT00]. Going one step further, to compiler design and construction, the classic “Dragon Book” [AHO86] is invaluable.