**Java Buzzwords**

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

**Simple:** Java was designed to be easy for the professional programmer to learn and use effectively. As users have some programming experience, it will not find Java hard to master. If users 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.

**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 primitive types, such as integers, are kept as high-performance nonobjects.

**Robust:**The multiplatformed 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. 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. The Java Language 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:** When Java was still a new language, it was criticized for being slow: Since Java bytecode was executed by an interpreter, it seemed that Java bytecode programs could never run as quickly as programs compiled into native machine language (that is, the actual machine language of the computer on which the program is running). However, this problem has been largely overcome by the use of just-in-time compilers (JIT) for executing Java bytecode.

JIT is a part of JVM. A just-in-time compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a just-in-time compiler is a Java bytecode program, and its task is to execute that program.

It is important to understand that it is not practical 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, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. The translated parts of the program can then be executed much more quickly than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a just-in-time compiler can significantly speed up the overall execution time.Even though dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

**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. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

**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 Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

**Security:** Every time when we download a “normal” program, we are are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) 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 innovative aspect of Java.

**Portability:** Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work on all computers. Therefore, some means of generating portable executable code was needed

**DataType:**

**Java Is a Strongly Typed Language:**

It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java’s safety and robustness comes from this fact. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

The Primitive Types Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:

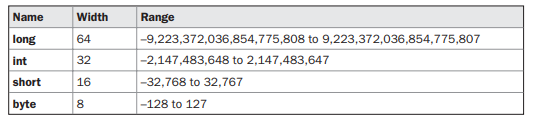
• **Integers** This group includes byte, short, int, and long, which are for whole-valued signed numbers.

• **Floating-point numbers** This group includes float and double, which represent numbers with fractional precision.

• **Characters** This group includes char, which represents symbols in a character set, like letters and numbers.

• **Boolean** This group includes boolean, which is a special type for representing true/false values.

Integers: Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java’s designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value. Java manages the meaning of the high-order bit differently, by adding a special “unsigned right shift” operator. Thus, the need for an unsigned integer type was eliminated. The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.



**byte**

The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you’re working with a stream of data from a network or file. They are also useful when you’re working with raw binary data that may not be directly compatible with Java’s other built-in types. Byte variables are declared by use of the byte keyword. For example, the following declares two byte variables called b and c: byte b, c;

**short**

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of short variable declarations: short s; short t;

**int**

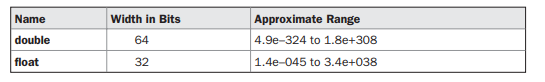
The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case. The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, int is often the best choice when an integer is needed.

**long**

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed.

**Floating-Point Types**

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, ortranscendentals such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:



**float**

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don’t require a large degree of precision. For example, float can be useful when representing dollars and cents.

Here are some example float variable declarations:

float hightemp, lowtemp;

**double**

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

**Characters**

In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

class Demo {

public static void main (String[]args) {

//Unicode escape sequence

char unicodeChar = '\u0041';

System.out.println("Stored Unicode Character: " + unicodeChar);

}

}

**boolean**

Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

class Demo {

public static void main (String[]args) {

boolean isJavaFun = true;

boolean isFishTasty = false;

System.out.println(isJavaFun); // Outputs true

System.out.println(isFishTasty); // Outputs false

}

}

**A First Simple Program**

. /\* This is a simple Java program. Call this file "Example.java". \*/

class Example

{

// Program begins with a call to main().

public static void main(String args[])

{

System.out.println("This is a simple Java program.");

}

}

Explanation:

The program begins with the following lines:

**/\* This is a simple Java program. Call this file "Example.java". \*/**

This is a comment. Like most other programming languages, Java lets you enter a remark into a program’s source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called Example.java

Java supports three styles of comments. The one shown at the top of the program is called a multiline comment. This type of comment must begin with /\* and end with \*/. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here:

**class Example {**

This line uses the keyword class to declare that a new class is being defined. Example is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). For the moment, don’t worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented.

The next line in the program is the single-line comment, shown here:

**// Your program begins with a call to main().**

This is the second type of comment supported by Java. A single-line comment begins with a // and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. The third type of comment, a documentation comment, will be discussed in the “Comments” section later in this chapter.

The next line of code is shown here:

**public static void main(String args[]) {**

This line begins the main( ) method.As the comment preceding it suggests, this is the line at which the program will begin executing.All Java applications begin execution by calling main( ).

The public keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared. (The opposite of public is private, which prevents a member from being used by code defined outside of its class.) In this case, main( ) must be declared as public, since it must be called by code outside of its class when the program is started. The keyword static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java Virtual Machine before any objects are made. The keyword void simply tells the compiler that main( ) does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don’t worry. All of these concepts will be discussed in detail in subsequent chapters.

As stated, main( ) is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, Main is different from main. It is important to understand that the Java compiler will compile classes that do not contain a main( ) method. But java has no way to run these classes. So, if you had typed Main instead of main, the compiler would The Java Language still compile your program. However, java would report an error because it would be unable to find the main( ) method. Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If there are no parameters required for a given method, you still need to include the empty parentheses. In main( ), there is only one parameter, albeit a complicated one. String args[ ] declares a parameter named args, which is an array of instances of the class String. (Arrays are collections of similar objects.) Objects of type String store character strings. In this case, args receives any command-line arguments present when the program is executed.

The last character on the line is the {. This signals the start of main( )’s body. All of the code that comprises a method will occur between the method’s opening curly brace and its closing curly brace. One other point: main( ) is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a main( ) method to get things started. When you begin creating applets—Java programs that are embedded in web browsers—you won’t use main( ) at all, since the web browser uses a different means of starting the execution of applets. The next line of code is shown here. Notice that it occurs inside main( ). System.out.println("This is a simple Java program."); This line outputs the string “This is a simple Java program.” followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. In this case, println( ) displays the string which is passed to it. As you will see, println( ) can be used to display other types of information, too. The line begins with System.out. System is a predefined class that provides access to the system, and out is the output stream that is connected to the console. As you have probably guessed, console output (and input) is not used frequently in most real-world Java programs and applets.println( ) statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements. The first } in the program ends main( ), and the last } ends the Example class definition.