Software Needed

All the software you will need for this class is available free for download from the web.

Java SE Software Development Kit (JDK)

Free Java Standard Edition Development Kit (JDK), which is available from:

Carefully follow the instructions for downloading and installing the software on your platform.

The downloads are available from:

We do not recommend modifying your system's CLASSPATH environment variable, which is discussed in the installation instructions.

Free versions of these tools can be downloaded from the web at the following sites:

- JDK : www.oracle.com/technetwork/java/javase/downloads/
- Eclipse IDE: eclipse.org/downloads/
- NetBeans IDE: netbeans.org/downloads/
- jGRASP IDE: spider.eng.auburn.edu/user-cgi/grasp/ grasp.pl?;dl=download_jgrasp.html
- DrJava IDE: drjava.org
- BlueJ IDE: www.bluej.org/download/download.html
- TextPad Text Editor (evaluation version): www.textpad.com/download

Introduction

Welcome to Java—the world's most widely used computer programming language. You're already familiar with the powerful tasks computers perform. You will now write instructions commanding computers to perform those kinds of tasks. Software (i.e., the instructions you write) controls hardware (i.e., computers). You will also learn object-oriented programming—today's key programming methodology (Possibly in COSC 406: Java programming II); where you will create and work with many software objects. Java is the preferred language for meeting many organizations' enterprise programming needs. Java has also become the language of choice for implementing Internet-based applications and software for devices that communicate over a network.

Java Editions: Standard Edition (SE), Enterprice Edition (EE) and Micro Edition (ME)

Java Standard Edition (Java SE)

The Java Enterprise Edition (Java EE)

Java EE is geared toward developing large-scale, distributed networking applications and web-based applications. In the past, most computer applications ran on "standalone" computers (computers that were not networked together). Today's applications can be written with the aim of communicating among the world's computers via the Internet and the web.

The Java Micro Edition (Java ME)

Java ME is geared toward developing applications for small, memory-constrained devices, such as BlackBerry smartphones. Google's Android operating system—used on numerous smartphones, tablets, e-readers and other devices—uses a customized version of Java not based on Java ME.

Machine Languages, Assembly Languages and High Level Languages:

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of such languages are in use today. These are divided into three general types:

1. Machine languages

2. Assembly languages

3. High-level languages

Any computer can directly understand only its own machine language, defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent (a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans.

Programming in machine language was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of assembly languages. Translator programs called assemblers were developed to convert early assembly-language programs to machine language at computer speeds. Although such code is clearer to humans, it is incomprehensible to computers until translated to machine language. Computer usage increased rapidly with the advent of assembly languages, but programmers still had to use many instructions to accomplish even the simplest tasks.

To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks. Translator programs called compilers convert high-level language programs into machine language. High-level languages allow you to write instructions that look almost like every-day English and contain commonly used mathematical notations. High-level languages are preferable to machine and assembly languages. Java is by far the most widely used high-level programming language.

Java and a Typical Java Development Environment

Microprocessors are having a profound impact in intelligent consumer-electronic devices. Recognizing this, Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in a C++-based object-oriented programming language Sun called Java. A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-control devices. This is sometimes called “write once, run anywhere.” The web exploded in popularity in 1993, and Sun saw the potential of using Java to add dynamic content, such as interactivity and animations, to web pages. Java garnered the attention of the business community because of the phenomenal interest in the web. Java is now used to develop large-scale enterprise applications, to enhance the functionality of web servers (the computers that provide the content we see in our web browsers), to provide applications for consumer devices (e.g., cell phones, smartphones, television set-top boxes and more) and for many other purposes. Java is now widely used software development language in the world.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer’s language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers. Because of the similarities between Java and C++, it is tempting to think of Java as simply the “Internet version of C++.” However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++

The Java Buzzwords

No discussion of Java’s history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

• Simple

• Secure

• Portable

• Object-oriented

• Robust

• Multithreaded

• Architecture-neutral

• Interpreted

• High performance

• Distributed

• Dynamic

Typical Java Development Environment

We now explain the commonly used steps in creating and executing a Java application using a Java development environment (illustrated in Figs. 1 - 5). Java programs normally go through five phases—edit, compile, load, verify and execute phases. Let’s discuss these stpes in the context of the Java SE Development Kit (JDK).

Phase 1: Creating a Program

Phase 1 consists of editing a file with an editor program, normally known simply as an editor (Fig. 1). You type a Java program (typically referred to as source code) using the editor, make any necessary corrections and save the program on a secondary storage device, such as your hard drive. A file name ending with the .java extension indicates that the file contains Java source code.

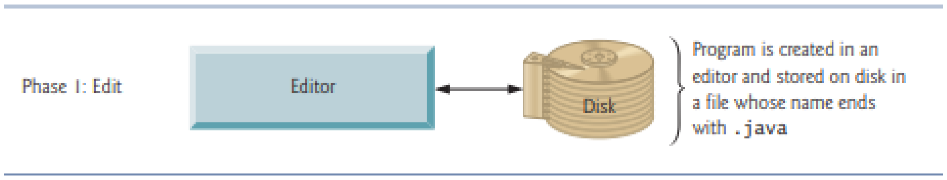

Phase 1: Edit
Editor
Disk
Program is created in an editor and stored on disk in a file whose name ends with .java

Figure 1:

Phase 2: Compiling a Java Program into Bytecodes

In Phase 2, you use the command javac (the Java compiler) to compile a program (Fig. 2). For example, to compile a program called Welcome.java, you would type in the command window of your system (i.e., the Command Prompt in Windows, the shell prompt in Linux or the Terminal application in Mac OS X).

```
Javac Welcome.java
```

If the program compiles, the compiler produces a .class file called Welcome.class that contains the compiled version of the program.

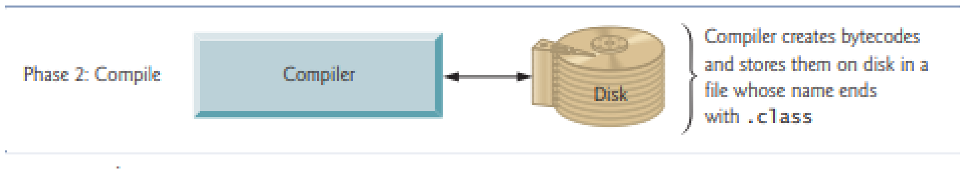

Phase 2: Compile
Compiler
Disk
Compiler creates bytecodes and stores them on disk in a file whose name ends with .class

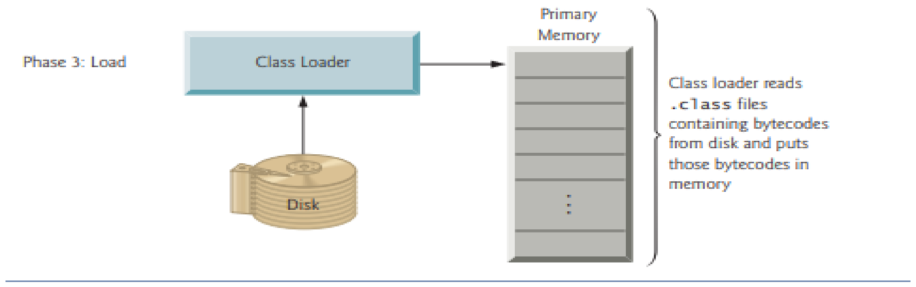
Figure 2:

The Java compiler translates Java source code into bytecodes that represent the tasks to execute in the execution phase (Phase 5). Bytecodes are executed by the Java Virtual Machine (JVM)—a part of the JDK and the foundation of the Java platform.

A virtual machine (VM) is a software application that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it. If the same VM is implemented on many computer platforms, applications that it executes can be used on all those platforms. The JVM is one of the most widely used virtual machines. Microsoft's .NET uses a similar virtual-machine architecture.

Phase 3: Loading a Program into Memory

In Phase 3, the JVM places the program in memory to execute it—this is known as loading (Fig. 3).The JVM's class loader takes the .class files containing the program's bytecodes and transfers them to primary memory. The class loader also loads any of the .class files provided by Java that your program uses. The .class files can be loaded from a disk on your system or over a network (e.g., your local college or company network, or the Internet).


Phase 3: Load
Class Loader
Primary Memory
Disk
Class loader reads .class files containing bytecodes from disk and puts those bytecodes in memory

Phase 4: Bytecode Verification

In Phase 4, as the classes are loaded, the bytecode verifier examines their bytecodes to ensure that they are valid and do not violate Java's security restrictions (Fig. 4). Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might)

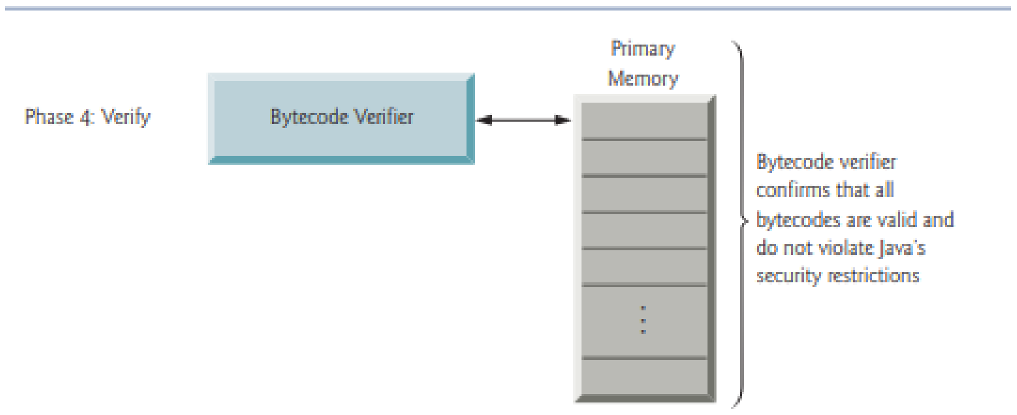

Phase 4: Verify
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions

Figure 4:

Phase 5: Execution

In Phase 5, the JVM executes the program's bytecodes, thus performing the actions specified by the program (Fig. 5). In early Java versions, the JVM was simply an interpreter for Java bytecodes. This caused most Java programs to execute slowly, because the JVM would interpret and execute one bytecode at a time. Some modern computer architectures can execute several instructions in parallel.

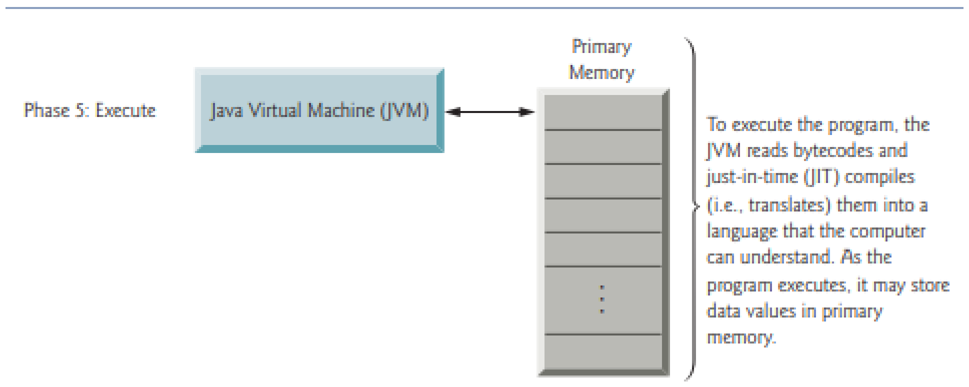

Phase 5: Execute
Java Virtual Machine (JVM)
Primary Memory
To execute the program, the JVM reads bytecodes and just-in-time (JIT) compiles (i.e., translates) them into a language that the computer can understand. As the program executes, it may store data values in primary memory.

Figure 5:

Thus, Java programs actually go through two compilation phases—one in which source code is translated into bytecodes (for portability across JVMs on different computer platforms) and a second in which, during execution, the bytecodes are translated into machine language for the actual computer on which the program executes.

A First Simple Program

Let us start by compiling and running the short sample program. As you will see, this involves a little more work than you might imagine.

```
/*      This is a simple Java program.
        Call this file "Example.java".
*/
 class Example {
        // Your program begins with a call to main().
        public static void main(String args[]) {
                System.out.println("This is a simple Java program.");
         }
}
```

Entering the Program

For most computer languages, the name of the file that holds the source code to a program is immaterial. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be Example.java.

As you can see by looking at the program, the name of the class defined by the program is also Example. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of the main class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program

To compile the Example program, execute the compiler, javac, specifying the name of the source file on the command line, as shown below:

```
C:\>javac Example.java
```

The javac compiler creates a file called Example.class that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of javac is not code that can be directly executed. To actually run the program, you must use the Java application launcher called java. To do so, pass the class name Example as a command-line argument, as shown below:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

A Closer Look at the First Sample Program Although Example.java is quite short, it includes several key features that are common to all Java programs. Let us closely examine each part of the program. 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.

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 (}). In Java, all program activity occurs within class. 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 below:

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

A single-line comment begins with a // and ends at the end of the line.

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 modifier*, which allows the programmer to control the visibility of class members.

The keyword *static* allows main( ) to be called without having to instantiate a particular instance of the class.

The keyword *void* simply tells the compiler that main( ) does not return a value.

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. 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.

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.

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.

Lexical Issues

Now that you have seen simple Java programs, it is time to more formally describe the atomic elements of Java. Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the Example program could have been written all on one line or in any other strange way you felt like typing it. In Java, whitespace is a space, tab, or newline.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of

- Uppercase and Lowercase Letters,
- Numbers,
- The Underscore
- Dollar-Sign Characters. (The Dollar-Sign Character Is Not Intended For General Use.)
- Must Not Begin With A Number
- Java is case-sensitive, so VALUE is a different identifier than Value.

Some examples of valid identifiers are

- COSC_315
- MyFirstProgram
- Reg_Number
- $Money

Some examples of invalid identifiers are:

- 2023Session
- COSC-315
- Month/Year
- My App

Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100

98.6

‘X’

“This is a test”

The first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Java Comments

Java supports three styles of comments

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.

Single-line Comment:

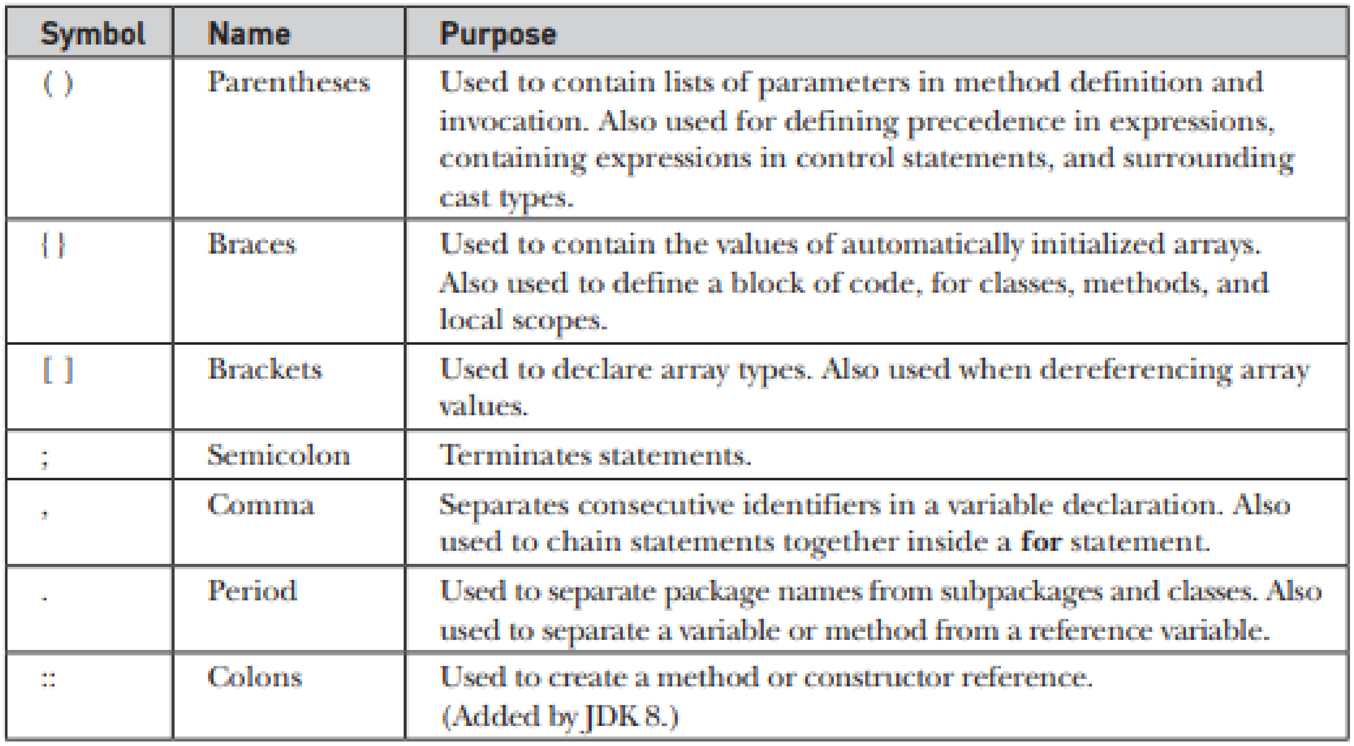
A single-line comment begins with a // and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions.

Documentation Comment:

This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */

Separators

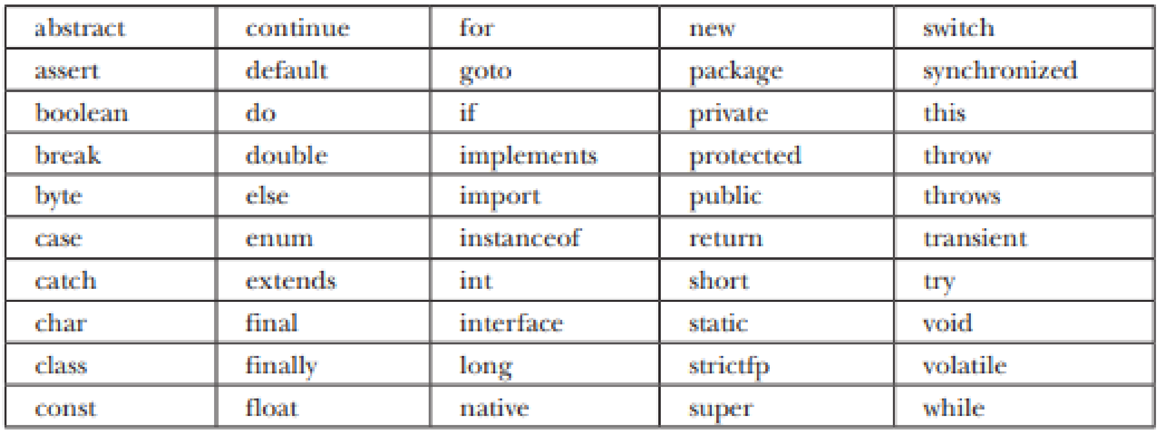
In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

The Java Keywords

The following Table contains keywords defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as identifiers. Thus, they cannot be used as names for a variable, class, or method.



abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

The Java Class Libraries The sample programs shown in this chapter make use of two of Java's built-in methods**: println( )** and **print( )**. These methods are available through **System.out.** System is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI). Thus, Java as a totality is a combination of the Java language itself, plus its standard classes.

Data Types

Java is a strongly typed language. Every variable has a type, every expression has a type, and every type is strictly defined. All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.

The Primitive Types

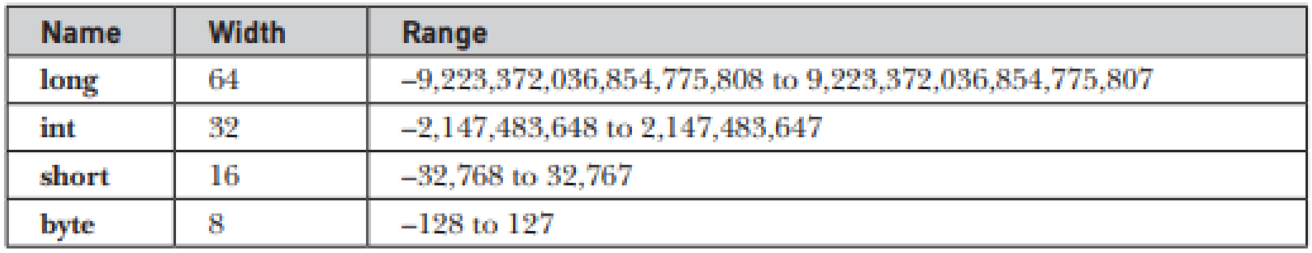
Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

The primitive types are also commonly referred to as simple types, and both terms will be used.

These can be put in four groups:

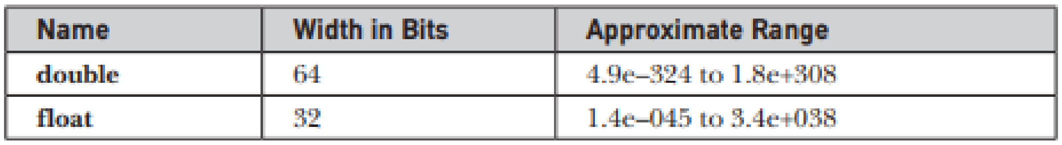
i. **Integers:** This group includes **byte, short, int, and long**, which are for whole-valued signed numbers.
ii. • **Floating-point numbers**: This group includes **float and double**, which represent numbers with fractional precision.
iii. • **Characters**: This group includes char, which represents symbols in a character set, like letters and numbers.
iv. • **Boolean** This group includes **boolean**, which is a special type for representing true/false values

The following Table gives the width and Range of **Integer** group:



Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

The following Table give the width and Range of the floating-point numbers:



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

In Java **char** is a 16-bit type. The range of a char is 0 to 65,536.

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier = value, identifier = value  …;
```

The following are some examples of variable declaration in Java:

int a, b, c; // declares three ints, **a, b,** and **c**.

int d = 3, e, f = 5; // declares three more ints, initializing // **d** and **f**.

byte z = 22; // initializes z.

double pi = 3.14159; // declares an approximation of pi.

char x = 'x'; // the variable x has the value 'x'.

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
// Demonstrate dynamic initialization.
    class DynInit {
        public static void main(String args[]) {
            double a = 3.0, b = 4.0;
            // c is dynamically initialized
            double c = Math.sqrt(a * a + b * b);
            System.out.println("Hypotenuse is " + c);
        }
    }
```

Here, three local variables—**a, b**, and **c**—are declared. The first two**, a** and **b**, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem).