

UNIT – I

Program Structure in Java: Introduction, Writing Simple Java Programs, Elements or Tokens in Java Programs, Java Statements, Command Line Arguments, User Input to Programs, Escape Sequences, Comments, Programming Style.

Data Types, Variables, and Operators: Introduction, Data Types in Java, Declaration of Variables, Data Types, Type Casting, Scope of Variable Identifier, Literal Constants, Symbolic Constants, Formatted Output with printf() Method, Static Variables and Methods, Attribute Final, Introduction to Operators, Precedence and Associativity of Operators, Assignment Operator (=), Basic Arithmetic Operators, Increment (++) and Decrement (--) Operators, Ternary Operator, Relational Operators, Boolean Logical Operators, Bitwise Logical Operators.

Control Statements: Introduction, if Expression, Nested if Expressions, if–else Expressions, Ternary Operator?., Switch Statement, Iteration Statements, while Expression, do–while Loop, for Loop, Nested for Loop, For–Each for Loop, Break Statement, Continue Statement.

1.1 Program Structure in Java

1.1.1 Introduction

The world is composed of objects such as stars, plants, trees, animals, fish, men, and women. Each name represents a group of objects that not only have some common characteristics among themselves but also have characteristics different from those on other groups.

Each object in this world is self-contained; in other words, it has its own methods of working and reacting to other objects. Thus, the world is driven by the interaction of these different types of objects. Similarly, an object-oriented program also works on the same concept. Here, the object and its methods are defined together in a class.

These classes are used in software programs that are meant to solve the problems of the real world. Therefore, different types of software objects can solve a physical problem by mutual interaction.

Here, we discuss the types of programs in Java, how to write them, and the types of problems in the real world that they can solve. In Java, there are three types of programs as follows:

- ✚ **Stand alone application programs:** These programs are made and run on users' computers.
- ✚ **Applet programs:** these programs are meant to run in a web page on the internet.
- ✚ **Java servlets:** These programs run in computers that provide web services. They are also often called server-side programs or servlets.

1.1.2 Writing Simple Java Programs

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. Let's see why. In Java, a source file is officially called a compilation unit. The Java compiler requires that a source file use the .java file name extension.

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.

Example:

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

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.

The contents of a comment are ignored by the compiler. 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 Sample {
```

This line uses the keyword class to declare that a new class is being defined. Sample 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 (}).

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. 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.

The **public** keyword is an *access modifier*, 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 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.

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.

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.

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**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

Notice that the **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 Sample class definition.

Compiling and Running Java Programs

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

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

The **javac** compiler creates a file called **Sample.class** that contains the byte code version of the program. As discussed earlier, the Java byte code 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.

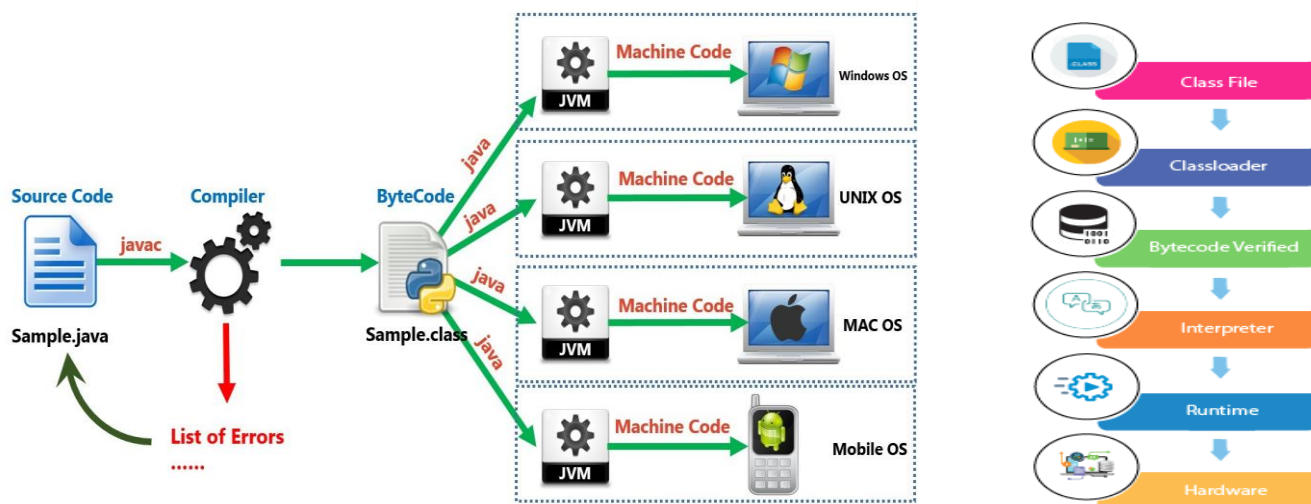
To do so, pass the class name Example as a command-line argument, as shown here:

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

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

This is a simple Java program.

When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the .class file. When you execute java as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the .class extension. If it finds the file, it will execute the code contained in the specified class.



1.1.3 Elements or Tokens in Java Programs

It is time to more formally describe the atomic elements (Tokens) 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 Sample program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator.

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, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a

number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Some examples of valid identifiers are

AvgTemp	count	a4	\$test	this_is_ok
---------	-------	----	--------	------------

Invalid identifier names include these:

2count	high-temp	Not/ok
--------	-----------	--------

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"
-----	------	-----	------------------

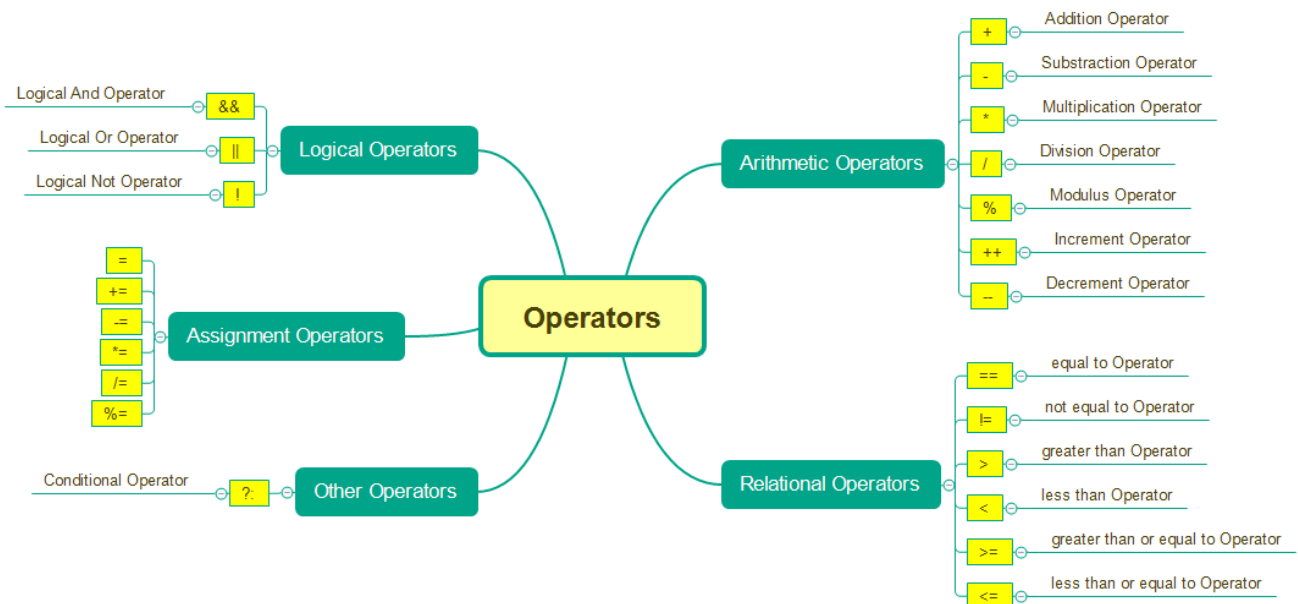
Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a 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 `*/`.

Operators

There are 37 operators in Java. Some of the important operators are given below.



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.

The Java Keywords

There are 50 keywords currently 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.

The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java defines only the keywords shown in below table.

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

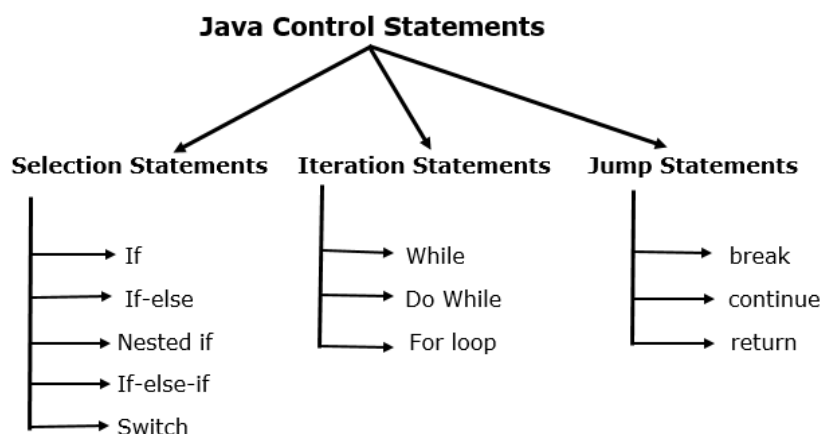
In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

1.1.4 Java Statements

The Java Program comprises a set of instructions; these instructions are the statements in Java. The statements specify the sequence of actions to be performed when some method or constructor is invoked. The details of these statements would be provided as and when required. The following are the some of the important java statements.

1. Empty statement
2. Variable declaration statement
3. Labelled statement
4. Expression statement
5. Control statements: Selection statement, Iteration statement, Jump statement

- 6. Synchronization statement (used with Multi-threading)
- 7. Guiding statement (used with Exception Handling)



1.1.5 Command Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to `main()`. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a String array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on.

For example, the following program displays all of the command-line arguments that it is called with:

// Display all command-line arguments.

```

class CommandLine {
public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " +args[i]);
    }
}
    
```

Try executing this program, as shown here:

```

java CommandLine this is a test 100 -1
    
```

When you do, you will see the following output:

```

args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
    
```

NOTE: All command line arguments are passed as strings. You must convert numeric values to their internal forms manually.

1.1.6 User Input to Programs

Here, we make use of the class Scanner of package java.util to give input to the program. Basically, Java Scanner class is a text scanner that breaks the input into using a delimiter. In Java program, importing the class Scanner from package is the first step and should be declare object for performing above stated action.

Example: Addition of two integer values

```
import java.util.Scanner; // importing Scanner class
public class Add {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in); // Declaring object
        int a, b;
        System.out.println("Enter any two integer values: ");
        a = sc.nextInt();    b = sc.nextInt(); // Here, sc invokes the method nextInt() which reads the value typed by the user.
        System.out.println("The Sum of given two integer values is : "+(a+b));
    }
}
```

1.1.7 Escape Sequences




These characters are consists of a backslash followed by a letter and have special meaning for the compiler. The character escape sequences are as follows:

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

In fact, these are the instructions to for manipulations in formatting and character representation. For example, “\u0041” is the representation of letter A.

1.1.8 Comments

The contents of a comment are ignored by the compiler. Java supports three styles of comments.

-  Single-line comment
-  Multi-line comment
-  Documentation comment

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

Multiline 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.

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 `*/`.

1.1.9 Programming Style

Java is a free form language. We need not have to indent any lines to make the program work properly. Java system does not care when on the line we begin typing. While this may be a license for bad programming style. We should try to use this fact to our advantage for producing readable programs. Although several alternate styles are possible, we should select one and try to use it with total consistency.

For example, the statement `System.out.println("Java is Wonderful!");`

can be written as

```
System.out.println
("Java is Wonderful!");
```

or, even as

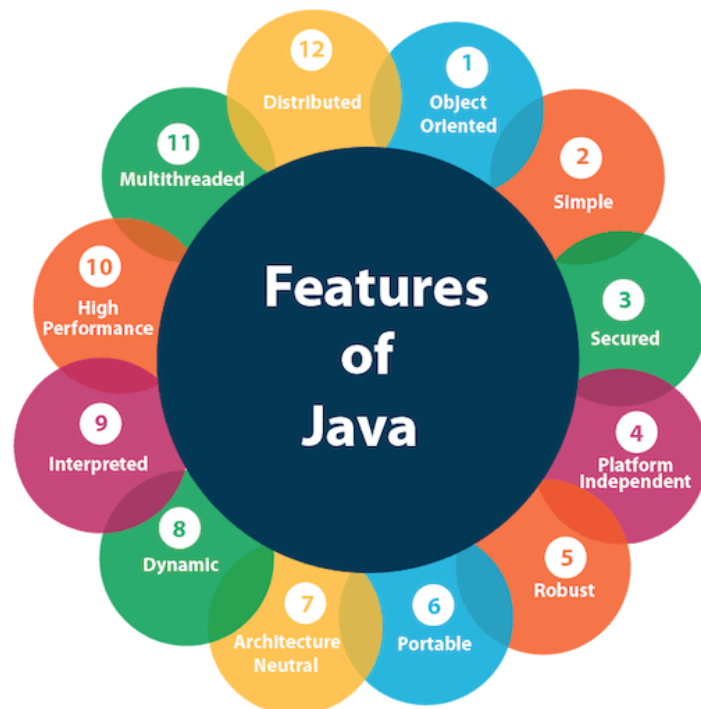
```
System.
out.println
("Java is Wonderful!"
);
```

Practise for good programming style:

- ✓ Appropriate comments
- ✓ Naming conventions
- ✓ Proper indentation and spacing lines
- ✓ Block styles

1.1.10 Additional Concept: Java Buzz Words

The key considerations were summed up by the Java team in the following list of buzzwords:



Object-oriented: A clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.

Simple: Java is designed to be easy for the professional programmer to learn and use.

Secured: programs are confined to the Java execution environment and cannot access other parts of the computer.

Robust: Restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exceptionhandling) checks and manages memory automatically.

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. The same mechanism that helps ensure security also helps create portability.
- Indeed, Java's solution to these two problems is both elegant and efficient.
-

Architecture-neutral: Java Virtual Machine provides a platform independent environment for the execution of Java byte code

Dynamic: substantial amounts of run-time type information to verify and resolve access to objects at run-time.

Interpreted and high-performance: Java programs are compiled into an intermediate representation – byte code:

- ✓ can be later interpreted by any JVM
- ✓ can be also translated into the native machine code for efficiency.

Multithreaded: Supports multi-threaded programming for writing program that perform concurrent computations.

Distributed: Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.

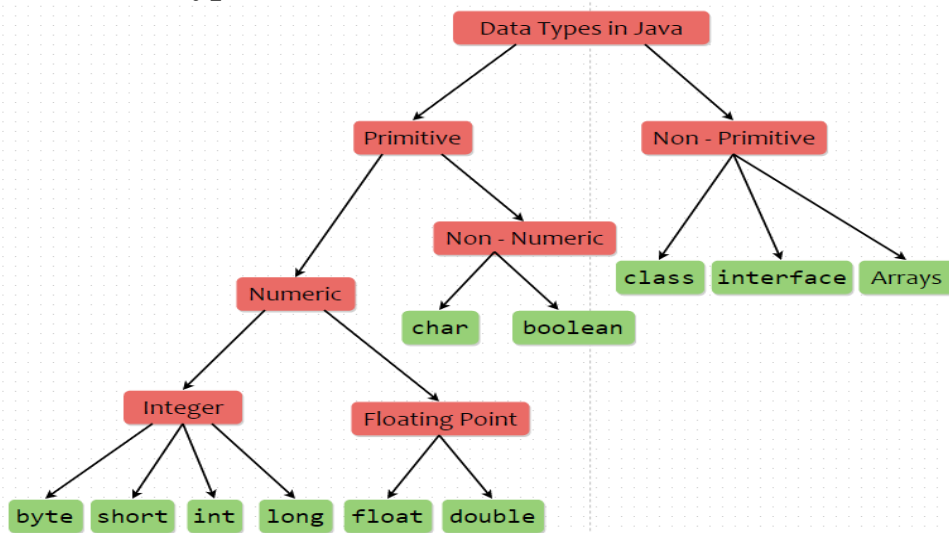
1. 2 Data Types, Variables, and Operators

1.2.1 Introduction

Java language programs deal with the following entities: Primitive Data, Classes and Objects, Interfaces and their references, Arrays, Methods.

The primitive data and their types are defined independent of the classes and interfaces. The arrays and methods derive their types from the first three entities.

1.2.2 Data Types in Java



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.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create. The primitive types represent single values—not complex objects.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java’s portability requirement, all data types have a strictly defined range. For example, an int is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

1.2.3 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. These elements are examined next.

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 *type* is one of Java's atomic types, or the name of a class or interface.

The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
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'.
```

The identifiers that you choose have nothing intrinsic in their names that indicate their type. Java allows any properly formed identifier to have any declared type.

1.2.4 Data Types

1.2.4.1 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.

As you will see later, Java manages the meaning of the highorder 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. The width and ranges of these integer types vary widely, as shown in this table:

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

Let's look at each type of integer.

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. Therefore, **int** is often the best choice when an integer is needed.

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.

1.2.4.2 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, or transcendental functions 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:

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

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.

1.2.4.3 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.

NOTE: In the formal specification for Java, **char** is referred to as an integral type, which means that it is in the same general category as **int**, **short**, **long**, and **byte**. However, because it's principal use is for representing Unicode characters, **char** is commonly considered to be in a category of its own.

1.2.4.4 Booleans

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**.

1.2.5 Type Casting

It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

1.2.5.1 Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

1.2.5.2 Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(target-type) value

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
    public static void main(String args[] ) {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);
        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When the **d** is converted to an **int**, its fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

1.2.5.3 Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision

required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a*b** is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store $50 * 2$, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast.

This is true even if, as in this particular case, the value being assigned would still fit in the target type. In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

1.2.5.4 The Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows: First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands are **double**, the result is **double**.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
}
}
```

Let's look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i/c**, **c** is promoted to **int**, and the result is of type **int**.

Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

1.2.6 The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main()** method. However, Java allows variables to be declared within any block. As explained in earlier, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope*. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java's strict, object-oriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it), when classes are described. For now, we will only examine the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. Although this book will look more closely at parameters in Chapter 6, for the sake of this discussion, they work the same as any other method variable.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.

This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class Scope {
public static void main(String args[]) {
    int x; // known to all code within main
    x = 10;
if(x == 10) { // start new scope
    int y = 20; // known only to this block
    // x and y both known here.
    System.out.println("x and y: " + x + " " + y);
    x = y * 2;
    }
    // y = 100; // Error! y not known here
    // x is still known here.
    System.out.println("x is " + x);
    }
}
```

As the comments indicate, the variable **x** is declared at the start of **main()**'s scope and is accessible to all subsequent code within **main()**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is only visible to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because **count** cannot be used prior to its declaration:

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program:

```
// Demonstrate lifetime of a variable.
class LifeTime {
public static void main(String args[]) {
    int x;
    for(x = 0; x < 3; x++) {
        int y = -1; // y is initialized each time block is entered
        System.out.println("y is: " + y); // this always prints -1
        y = 100;
        System.out.println("y is now: " + y);
    }
}
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is reinitialized to **-1** each time the inner **for** loop is entered. Even though it is subsequently assigned the value **100**, this value is lost.

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
public static void main(String args[]) {
    int bar = 1;
    {
        // creates a new scope
        int bar = 2; // Compile-time error - bar already defined!
    }
}
}
```

1.2.7 Literal Constants

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. There are two other bases which can be used in integer literals, *octal* (base eight) and *hexadecimal* (base 16).

Beginning with JDK 7, you can also specify integer literals using binary. To do so, prefix the value with **0b** or **0B**. For example, this specifies the decimal value 10 using a binary literal:

```
int x = 0b1010;
```

Among other uses, the addition of binary literals makes it easier to enter values used as bitmasks. In such a case, the decimal (or hexadecimal) representation of the value does not visually convey its meaning relative to its use. The binary literal does.

Also beginning with JDK 7, you can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

```
int x = 123_456_789;
```

the value given to **x** will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits.

For example, this is valid:

```
int x = 123__456__789;
```

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. They are also useful for providing visual groupings when specifying binary literals.

For example, binary values are often visually grouped in four-digits units, as shown here:

```
int x = 0b1101_0101_0001_1010;
```

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation.

Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.

For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.

Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by a decimal number, which can be positive or negative.

Examples include 6.022E23, 314159E-05, and 2e+100.

Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant. You can also explicitly specify a **double** literal by appending a *D* or *d*. Doing so

is, of course, redundant. The default **double** type consumes 64 bits of storage, while the smaller **float** type requires only 32 bits.

Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a **P** or **p**, rather than an **E** or **e**, is used. For example, `0x12.2P2` is a valid floating-point literal. The value following the **P**, called the *binary exponent*, indicates the power-of-two by which the number is multiplied. Therefore, `0x12.2P2` represents 72.5.

Beginning with JDK 7, you can embed one or more underscores in a floating-point literal. This feature works the same as it does for integer literals, which were just described.

Its purpose is to make it easier to read large floating-point literals. When the literal is compiled, the underscores are discarded. For example, given

```
double num = 9_423_497_862.0;
```

the value given to **num** will be 9,423,497,862.0. The underscores will be ignored. As is the case with integer literals, underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. It is also permissible to use underscores in the fractional portion of the number. For example, `double num = 9_423_497.1_0_9;` is legal. In this case, the fractional part is **.109**.

Boolean Literals

Boolean literals are simple. There are only two logical values that a **boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation.

The **true** literal in Java does not equal 1, nor does the **false** literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as **boolean** or used in expressions with Boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as `'a'`, `'z'`, and `'@'`. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as `'\"` for the single-quote character itself and `'\n'` for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, `'\141'` is the letter `'a'`. For hexadecimal, you enter a backslash-u (`\u`), then exactly four hexadecimal digits. For example, `'\u0061'` is the ISO-Latin-1 `'a'` because the top byte is zero. `'\ua432'` is a Japanese Katakana character.

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

```
"Hello World"
```

```
"two\nlines"
```

```
" \"This is in quotes\""
```

1.2.8 Symbolic Constants

A symbolic constant is a variable whose value does not change throughout the program. Some of the examples include PI, NORTH, and EAST etc.

It is usually preferred to declare the symbolic constants using all the capital letters in a program as follows:

```
public static final PI = 3.145926535;
public static final C = 299792458; // speed of light
```

1.2.9 Formatted Output with printf() Method

In Java, the formatting of output may be carried out in two ways:

- By using class Formatter
- By method printf()

The formatter class is discussed later. Now we discuss the printf method that has been adapted by Java from programming language C and modified and upgraded.

Examples:

```
System.out.printf("Price of this item is ",22,"Rupees");
System.out.printf("%d %f\t%c %s \n", n, x, ch. str);
System.out.printf("%X \n", 163);
System.out.printf("%o \n", 163);
```

Formatting Output of Integer Numbers

Field width and left and right justification the field width is set by integer number placed between % sign and conversion character. The default justification is the right justification. For making it left justification, we place – sign after the % sign and before the field width number.

```
int p = 73;
```

```
System.out.printf("%d \n",p);
System.out.printf("|%20d\n",p);
System.out.printf("|%-20d\n,p);
```

Output for the above statements:

```
73
|          73|
|73          |
```

Formatting Output of Floating Point Numbers

The following of floating point numbers involve the following three parameters:

- *Field width*: It is specified by an integer value between the % sign and the conversion letter f or F
- *Justification*: Right justification is default. The left justification is achieved by a negative sign after % sign.
- *Precision*: The number of digits after the decimal point is specified by introducing a period followed by a number after % symbol. For example, .3 specifies three digits after the decimal point.

1.2.12 Introduction to Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types. The unary minus operator negates its single operand. The unary plus operator simply returns the value of its operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

This version uses the **+=** *compound assignment operator*. Both statements perform the same action: they increase the value of **a** by 4.

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

In this case, the %= obtains the remainder of $a / 2$ and puts that result back into **a**. There are compound assignment operators for all of the arithmetic, binary operators.

Thus, any statement of the form
`var = var op expression;`

can be rewritten as
`var op= expression;`

The compound assignment operators provide two benefits. First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms. Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms. For these reasons, you will often see the compound assignment operators used in professionally written Java programs.

Here is a sample program that shows several `op=` assignments in action:

```
// Demonstrate several assignment operators.
```

```
class OpEquals {
public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement

The ++ and the -- are Java’s increment and decrement operators. Here they will be discussed in detail. As you will see, they have some special properties that make them quite interesting. Let’s begin by reviewing precisely what the increment and decrement operators do.

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in *postfix* form, where they follow the operand as just shown, and *prefix* form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.

For example:

```
x = 42;  
y = ++x;
```

In this case, **y** is set to 43 as you would expect, because the increment occurs *before* **x** is assigned to **y**. Thus, the line **y = ++x;** is the equivalent of these two statements:

```
x = x + 1;  
y = x;
```

However, when written like this,

```
x = 42;  
y = x++;
```

the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. Of course, in both cases **x** is set to 43. Here, the line **y = x++;** is the equivalent of these two statements:

```
y = x;  
x = x + 1;
```

The following program demonstrates the increment operator.

```
// Demonstrate ++.  
class IncDec {  
public static void main(String args[]) {  
    int a = 1;  
    int b = 2;  
    int c;  
    int d;  
        c = ++b;  
        d = a++;  
        c++;  
    System.out.println("a = " + a);  
    System.out.println("b = " + b);  
    System.out.println("c = " + c);  
    System.out.println("d = " + d);  
    }  
}
```

The output of this program follows:

```
a = 2  
b = 3  
c = 4  
d = 1
```

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The Assignment Operator

Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, =.

The assignment operator works in Java much as it does in any other computer language. It has this general form: *var = expression*;

Here, the type of *var* must be compatible with the type of *expression*. The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments.

For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-thenelse statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

expression1 ? expression2 : expression3

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the ? operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same (or compatible) type, which can't be **void**.

Here is an example of the way that the ? is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the *left* of the question mark. If **denom** equals zero, then the expression *between* the question mark and the colon is evaluated and used as the value of the entire ? expression. If **denom** does not equal zero, then the expression *after* the colon is evaluated and used for the value of the entire ? expression. The result produced by the ? operator is then assigned to **ratio**.

Here is a program that demonstrates the ? operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
public static void main(String args[]) {
    int i, k;
    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
}
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the [], (), and . can also act like operators. In that capacity, they would have the highest precedence.

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
=	op=					
Lowest						

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire.

For example, consider the following expression:

```
a >> b + 3
```

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

```
a >> (b + 3)
```

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a >> b) + 3
```

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later.

For example, which of the following expressions is easier to read?

```
a | 4 + c >> b & 7
```

```
(a | (((4 + c) >> b) & 7))
```

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

1.3 Control Statements

1.3.1 Introduction

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program.

Java's program control statements can be put into the following categories: selection, iteration, and jump.

- ✓ **Selection** statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- ✓ **Iteration** statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- ✓ **Jump** statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

1.3.2 Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

if

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

```
if (condition) statement1;  
else statement2;
```

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed.

For example, consider the following:

```
int a, b;  
//...  
if(a < b) a = 0;  
else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Nested ifs

A *nested if* is an **if** statement that is the target of another **if** or **else**. Nested **ifs** are very common in programming. When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```
if(i == 10) {  
  if(j < 20) a = b;  
  if(k > 100) c = d; // this if is  
  else a = c; // associated with this else  
}
```


else a = d; // this else refers to if(i == 10)

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **ifs** is the *if-elseif* ladder. It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Here is a program that uses an if-else-if ladder to determine which season a particular month is in.

// Demonstrate if-else-if statements.

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}
}
```

Here is the output produced by the program:

April is in the Spring.

You might want to experiment with this program before moving on. As you will find, no matter what value you give month, one and only one assignment statement within the ladder will be executed.

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN :
// statement sequence
break;
default:
// default statement sequence
}
```

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

```
switch(count) {
case 1:
switch(target) { // nested switch
case 0:
System.out.println("target is zero");
break;
case 1: // no conflicts with outer switch
System.out.println("target is one");
break;
}
break;
case 2: // ...
```

Here, the **case 1:** statement in the inner switch does not conflict with the **case 1:** statement in the outer switch. The **count** variable is compared only with the list of cases at the outer level. If **count** is 1, then **target** is compared with the inner list cases.

In summary, there are three important features of the **switch** statement to note:

- ✓ The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- ✓ No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- ✓ A **switch** statement is usually more efficient than a set of nested **ifs**.

The last point is particularly interesting because it gives insight into how the Java compiler works.

1.3.3 Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {  
// body of loop  
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while loop.  
class While {  
public static void main(String args[]) {  
int n = 10;  
while(n > 0) {  
System.out.println("tick " + n);  
n--;  
}  
}  
}
```

When you run this program, it will "tick" ten times:

```
tick 10  
tick 9  
tick 8  
tick 7  
tick 6  
tick 5  
tick 4  
tick 3  
tick 2  
tick 1
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println()** is never executed:

```
int a = 10, b = 20;  
while(a > b)  
System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java's loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```
// The target of a loop can be empty.  
class NoBody {  
public static void main(String args[]) {  
int i, j;
```

```

i = 100;
j = 200;
// find midpoint between i and j
while(++i < --j); // no body in this loop
System.out.println("Midpoint is " + i);
}
}

```

This program finds the midpoint between **i** and **j**. It generates the following output: Midpoint is 150

Here is how this **while** loop works. The value of **i** is incremented, and the value of **j** is decremented. These values are then compared with one another. If the new value of **i** is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than **j**, the loop stops. Upon exit from the loop, **i** will hold a value that is midway between the original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Its general form is

```

do {
// body of loop
} while (condition);

```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Here is a reworked version of the "tick" program that demonstrates the **do-while** loop. It generates the same output as before.

// Demonstrate the do-while loop.

```

class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
n--;
} while(n > 0);
}
}

```

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```

do {
System.out.println("tick " + n);
} while(--n > 0);

```

In this example, the expression (**--n > 0**) combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **--n** statement executes, decrementing **n** and returning the

new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise, it terminates.

for

You were introduced to a simple form of the **for** loop. As you will see, it is a powerful and versatile construct.

Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new “for-each” form. Both types of **for** loops are discussed here, beginning with the traditional form.

Here is the general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {  
// body  
}
```

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is executed only once. Next, *condition* is evaluated. This must be a Boolean expression.

It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

The For-Each Version of the for Loop

Beginning with JDK 5, a second form of **for** was defined that implements a “for-each” style loop. As you may know, contemporary language theory has embraced the for-each concept, and it is quickly becoming a standard feature that programmers have come to expect. A foreach style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of **for** is also referred to as the *enhanced for* loop.

The general form of the for-each version of the **for** is shown here:

```
for(type itr-var : collection) statement-block
```

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. (Other types of collections that can be used with the **for**, such as those defined by the Collections Framework, are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the element type of the array.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace.

The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end.

For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<10; i++) {
for(j=i; j<10; j++)
System.out.print(".");
System.out.println();
}
}
}
```

The output produced by this program is shown here:

```
.....
.....
.....
.....
.....
.....
.....
....
...
..
.
```

1.3.4 Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here.

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of **goto**. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a **goto** just past the body of the loop, to the loop’s end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
public static void main(String args[]) {
```



```
    for(int i=0; i<10; i++) {
        System.out.print(i + " ");
        if (i%2 == 0) continue;
    System.out.println(" ");
    }
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in Chapter 6, a brief look at **return** is presented here.

At any time in a method the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main()**:

// Demonstrate return.

```
class Return {
public static void main(String args[]) {
    boolean t = true;
        System.out.println("Before the return.");
    if(t) return; // return to caller
        System.out.println("This won't execute.");
    }
}
```

The output from this program is shown here:

Before the return.

UNIT II:

Classes and Objects: Introduction, Class Declaration and Modifiers, Class Members, Declaration of Class Objects, Assigning One Object to Another, Access Control for Class Members, Accessing Private Members of Class, Constructor Methods for Class, Overloaded Constructor Methods, Nested Classes, Final Class and Methods, Passing Arguments by Value and by Reference, Keyword this.

Methods: Introduction, Defining Methods, Overloaded Methods, Overloaded Constructor Methods, Class Objects as Parameters in Methods, Access Control, Recursive Methods, Nesting of Methods, Overriding Methods, Attributes Final and Static.

2.1 .1 Classes Introduction:

Class is a user defined new data type. Once defined, this new type can be used to create objects of that type.

Class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably

The General Form of a Class:

A class is declared by use of the class keyword.

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;
    type methodname1(parameter-list)
    {
    // body of method
    }
    type methodname2(parameter-list)
    {
    // body of method
    }
    // ...
    type methodnameN(parameter-list)
    {
    // body of method
    }
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that de can be used.

A Simple Class and its members

```
class Box
{
double width;
double height;
double depth;
void volume()
{
```

```
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

Above class contains members as width, height, depth of double type and member function volume().

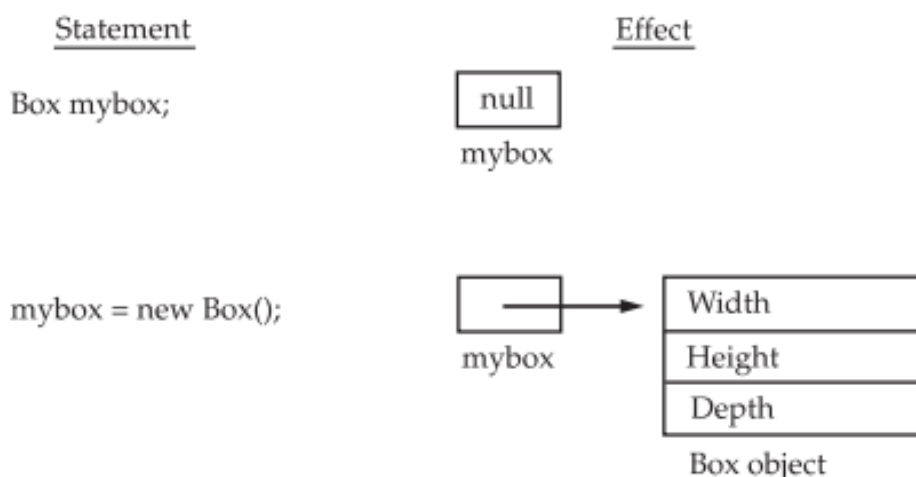
2.1.2 Declaring Objects:

Declaring objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

Ex: Box mybox = new Box();

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object

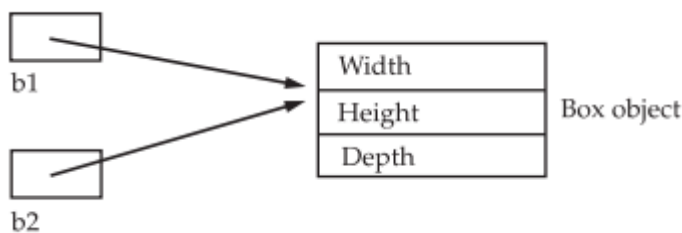


2.1.3 Assigning One Object to Another:

Box b1 = new Box();
Box b2 = b1;

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong.

Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.



2.1.4 Access Control for Class Members:

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control.

Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages.

Java’s access specifiers are **public, private, and protected.**

Public:

When a member of a class is modified by the public specifier, then that member can be accessed by any other code

Private:

When a member of a class is specified as private, then that member can only be accessed by other members of its class.

Protected:

When a member of a class is specified as protected, then that member can be accessed by its sub classes with in the package

When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

Access Modifiers In Java

Access Modifier	Within the Class	Other Classes [Within the Package]	In Subclasses [Within the package and other packages]	Any Class [In Other Packages]
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Same Package – Y Other Packages - N	N
private	Y	N	N	N

Y – Accessible
N – Not Accessible

2.1.5 Accessing Private Members of Class:

```

class Test {
    int a;    // default access
    public int b; // public access
    private int c; // private access
    // methods to access c
    void setc(int i) {
        c = i;
    }
    int getc() {
        return c;
    }
}
public class Main {
    public static void main(String args[]) {
        Test ob = new Test();
        ob.a = 1;
        ob.b = 2;
        // This is not OK and will cause an error
        // ob.c = 100; // Error!
        // You must access c through its methods
        ob.setc(100); // OK
        System.out.println("a, b, and c: " + ob.a +
            " " + ob.b + " " + ob.getc());
    }
}

```

2.1.6 Constructor Methods for Class:

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately

```

class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // compute and return volume

```

```

double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}

```

When this program is run, it generates the following results:

```

Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0

```

Types of java constructors :

There are two types of constructors:

- Default constructor (no-arg constructor)
- Parameterized constructor

//Default constructor

```

class Bike1
{
Bike1()
{
System.out.println("Bike is created");
}
public static void main(String args[])
{
Bike1 b=new Bike1();
}
}

```

//Parameterized constructor

```

class Student4
{
int id;
String name;
Student4(int i,String n)
{
id = i;
name = n;
}
}

```

```
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
Student4 s1 = new Student4(111,"Karan");
Student4 s2 = new Student4(222,"Aryan");
s1.display();
s2.display();
}
}
```

2.1.7 Overloaded Constructor Methods:

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the Box class developed in the preceding chapter. Following is the latest version of Box:

```
class Box
{
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume()
{
return width * height * depth;
}
}
```

Nested classes:

Demonstrate an inner class.

```
class Outer
{
int outer_x = 100;
void test()
{
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
```



```

void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo
{
public static void main(String args[])
{
Outer outer = new Outer();
outer.test();
}
}

```

Output from this application is shown here:
display: outer_x = 100

In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. An instance method named display() is defined inside Inner. This method displays outer_x on the standard output stream. The main() method of InnerClassDemo creates an instance of class Outer and invokes its test() method. That method creates an instance of class Inner and the display() method is called.

2.1.8 Final Class and Methods:

Final Class:

When a class is declared with final keyword, it is called a final class. A final class cannot be extended(inherited).

There are two uses of a final class :

1. One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer,Float etc. are final classes. We can not extend them.

```

final class A
{
// methods and fields
}
// The following class is illegal.
class B extends A
{
// COMPILE-ERROR! Can't subclass A
}

```

2. The other use of final with classes is to create an immutable class like the predefined String class. You cannot make a class immutable without making it final.

Final Method:

When a method is declared with final keyword, it is called a final method.

A final method cannot be overridden.

The Object class does this—a number of its methods are final. We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. The following fragment illustrates final keyword with a method:

```
class A
{
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

2.1.9 Passing Arguments by Value and by reference:

In general, there are two ways that a computer language can pass an argument to a subroutine.

The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

The second way an argument can be passed is call-by-reference.

In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed.

For example, consider the following program:

// Simple types are passed by value.

```
class Test
{
    void meth(int i, int j)
    {
        i *= 2;
        j /= 2;
    }
}

class CallByValue
{
    public static void main(String args[])
    {
        Test ob = new Test();
        int a = 15, b = 20;
```

```

System.out.println("a and b before call: " + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " + a + " " + b);
}
}

```

The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

Pass by Reference: It is a process in which the actual copy of reference is passed to the function. This is called by Reference.

Talking about Java, we can say that Java is Pass by Value and not pass by reference.

2.1.10 this Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the `this` keyword. `this` can be used inside any method to refer to the current object. That is, `this` is always a reference to the object on which the method was invoked. You can use `this` anywhere a reference to an object of the current class is needed. To consider the following version of `Box()`:

```

// A redundant use of this.
Box(double w, double h, double d)
{
this.width = w;
this.height = h;
this.depth = d;
}

```

This version of `Box()` operates exactly like the earlier version. The use of `this` is redundant, but perfectly correct. Inside `Box()`, `this` will always refer to the invoking object. While it is redundant in this case, `this` is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding:

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which hide instance variables. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.

```

// Use this to resolve name-space collisions.
Box(double width, double height, double depth)
{
this.width = width;
this.height = height;
this.depth = depth;
}

```

A word of caution: The use of `this` in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables.

2.2.1 Introducing Methods:

This is the general form of a method:

```
type name(parameter-list)
{
// body of method
}
```

type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void.

The name of the method is specified by name. This can be any legal identifier other than those already used by other items within the current scope.

The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than void return a value to the calling routine using the following form of the return statement: return value; Here, value is the value returned

```
class Box
{
double width;
double height;
double depth;
// display volume of a box
void volume()
{
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

```
class BoxDemo3
{
public static void main(String args[])
{
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
```

```
}  
}
```

Returning a Value:

While the implementation of `volume()` does move the computation of a box's volume inside the `Box` class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement `volume()` is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}  
class BoxDemo4 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
        // assign values to mybox1's instance variables  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;  
        /* assign different values to mybox2's  
        instance variables */  
        mybox2.width = 3;  
        mybox2.height = 6;  
        mybox2.depth = 9;  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

2.2.2 Overloading Methods:

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as **method overloading**. Method overloading is one of the ways that Java implements polymorphism.

// Demonstrate method overloading.

```
class OverloadDemo {  
    void test() {
```

```

System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}

```

This program generates the following output:

No parameters a:

10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

As you can see, test() is overloaded four times.

2.2.3 Class Objects as Parameters in methods:

```

class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {

```

```

public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}

```

This program generates the following output:

ob1 == ob2: true

ob1 == ob3: false

As you can see, the equals() method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns true. Otherwise, it returns false. Notice that the parameter o in equals() specifies Test as its type

2.2.4 Recursive Methods:

Java supports recursion. Recursion is the process of defining something in terms of itself.

As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

// A simple example of recursion(factorial).

```

class Factorial {
// this is a recursive function
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}

```

The output from this program is shown here:

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

2.2.5 Nesting of methods:

A method of a class can be called only by an object of that class using the dot operator. So, there is an exception to this. A method can be called by using only its name by another method of the same class that is called Nesting of Methods.

```

class test
{

```



```

int a,b;
test(int p, int q)
{
a=p;
b=q;
}
int greatest()
{
if(a>=b)
return(a);
else
return(b);
}
void display()
{
int great=greatest();
System.out.println("The Greatest Value="+great);
}
}
class temp
{
public static void main(String args[])
{
test t1=new test(25,24);
t1.display();
}
}

```

2.2.6 Method overriding:

- In a class hierarchy, when a method in a sub class has the same name and type signature as a method in its super class, then the method in the sub class is said to be override the method in the sub class.
- When an overridden method is called from within a sub class, it will always refers to the version of that method defined by the sub class
- The version of the method defined in the super class is hidden.
- In this situation, first it checks the method is existed in super class are not. If it is existed then it executes the version of sub class otherwise it gives no such method found exception.

Note: Methods with different signatures overloading but not overriding.

// Method overriding.

```

class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {

```

```
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
Output:
k: 3
```

Dynamic method dispatch:

- It is a mechanism by which a call to an overridden method is resolved at run time rather than compile time.
- It is important because this is how java implements runtime polymorphism.
- Before going to that we must know about super class reference sub class object.

Example:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
```

```

r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
callme(); // calls C's version of callme
}
}

```

Output:

```

Inside A's callme method
Inside B's callme method
Inside C's callme method

```

2.2.7 Attributes Final and Static

Final:

The modifier final may be used with variables as well as methods and classes. If a variable is declared as final it makes it constant. Its value cannot be changed in a program. The value is assigned right in the beginning.

Example:

```

Class PhyTest
{
Final int Minmarks=35; // final variable declaration
Void run()
{
Minmarks=40; //Value of final variable changed ,error
System.out.println("The subject is cleared");

Public static void main(String args[])
{
PhyTest obj=new PhyTest();
Obj.run();
}
}

```

Since the variable Minmarks is declared as final it cannot be modified throughout the program

Static Methods:

By declaring a method as static, it becomes a class method. Then it is not necessary that it should be called through an object. Static methods belong to the class and therefore, it can be accessed without declaring an object of the class.

Example:

```

Class Computers
{
Public static void main(String args[])
{
Show(); //calling method without an object
Computers obj=new Computers();
Obj.display(); //calling a method using an object
}
Static void show()
{

```

```
System.out.println("components details of computer");
}
Void display()
{
System.out.println("Main components of computer");
}
}
```

Note: static methods can access only static variables