MODULE 1

CHAPTER 1: INTRODUCTION

- Java was invented by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995.
- The primary motivation was the need for a platform-independent (that is, architecture-neutral)language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.
- With the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs. By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.
- Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, **Java** is a

programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives the programmer full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

- **While it is true that Java was influenced by C++, it is not an enhanced version of C++.**

BYTECODE

- The output of a Java compiler is not executable code. Rather, it is bytecode. ***Bytecode*** is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM).*
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.**The fact that a Java program is executed by the JVM also helps to make it secure.** Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.
- In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code ,but in java the differential between the two is not so great because of highly optimized bytecode. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis.

The Java Buzzwords

• Simple
• Secure
• Portable
• Object-oriented
• Robust
• Multithreaded
• Architecture-neutral

- Interpreted
- High performance
- Distributed
- Dynamic

Simple

If basic concepts of object-oriented programming is known and if a person is experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++.

Object-Oriented

The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in programdevelopment. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

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

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

Architecture-Neutral

Acentral issue for the Java designers was that of code longevity and portability. One of themain problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "**write once; run anywhere, any time, forever."**

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compilinginto an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

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

Object-Oriented Programming

Everything in java is an object. Object oriented means organizing the software as a combination of different types of object that incorporates both data and behavior. OOPs is a methodology that simplifies software development and maintenance by providing some rules.

Two Paradigms

All computer programs consist of two elements: code and data. These are the two paradigms that govern how a program is constructed. The first way is called the ***process-oriented model***. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data.* Procedural languages such as C employ this model.

The second approach, called ***object-oriented programming***, was conceived. Object-oriented programming organizes a program around its data (that is,objects) and a set of well-defined interfaces to that data. An object-oriented program can becharacterized as *data controlling access to code.*

Abstraction

Hiding implementation details from user and providing only the functionality.

The Three OOP Principles

1. **Encapsulation**

   Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
2. **Inheritance**
3. **Polymorphism**

A First Simple Program

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

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

Save the filename with classname. Here it is Example.java

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

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

The **javac**compiler creates a file called **Example.class**that contains the bytecode version of the program.

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.
```

Example 2:

```
/*
Here is another short example.
Call this file "Example2.java".
*/
class Example2 {
public static void main(String args[]) {
intnum; // this declares a variable called num
num = 100; // this assigns num the value 100
System.out.println("This is num: " + num);
num = num * 2;
System.out.print("The value of num * 2 is ");
```

```
System.out.println(num);
}
}
```

When you run this program, you will see the following output:

This is num: 100

The value of num * 2 is 200

Two Control Statements

The if Statement

Syntax:

if(*condition*) *statement*;

Here, *condition* is a Boolean expression. If *condition* is true, then the statement is executed. If *condition* is false, then the statement is bypassed. Here is an example:

```
if(num< 100) System.out.println("num is less than 100");
```

In this case, if num contains a value that is less than 100, the conditional expression is true, and println( ) will execute. If num contains a value greater than or equal to 100, then theprintln( ) method is bypassed

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

Program that illustrates the if statement:

```
/*
Demonstrate the if.
Call this file "IfSample.java".
*/
classIfSample {
```

```
public static void main(String args[]) {
int x, y;
x = 10;
y = 20;
if(x < y) System.out.println("x is less than y");
x = x * 2;
if(x == y) System.out.println("x now equal to y");
x = x * 2;
if(x > y) System.out.println("x now greater than y");
// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

The for Loop

Syntax:

for(*initialization; condition; iteration*) *statement*;

The ***initialization*** portion of the loop sets a loop control variable to an initial value. The ***condition*** is a Boolean expression that tests the loop control variable.

If the outcome of that test is true, the **for**loop continues to iterate. If it is false, the loop terminates. The ***iteration*** expression determines how the loop control variable is changed each time the loop iterates.

Program that illustrates the forloop:

```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
classForTest {
```

```
public static void main(String args[]) {
int x;
for(x = 0; x<10; x = x+1)
System.out.println("This is x: " + x);
}
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

By use of the increment operator, the preceding statement can be written like this:

```
x++;
```

Lexical Issues

1. **Whitespace**

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. Java is a free-form language. This means that you do not need to follow any special indentation

rules.

In Java, whitespace is a space, tab, or newline

2. **Identifiers**

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number. Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Valid identifiers are

AvgTemp count a4 $test this_is_ok

Invalid identifier

2count high-temp Not/ok

3. Literals

Aconstant 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"

4. Comments

There are three types of comments defined by Java

1. Single line
2. Multiline
3. Documentation: This type of comment is used to produce an HTMLfile that documents your program. The documentation comment begins with a /** and ends with a */.

5. Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon.

Symbol	Name	Purpose
( )	**Parentheses**	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expression 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 Primitive Types

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

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

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

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

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

• Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positiveand negative values.

Name	Width	Range
long	64	–9,223,372,036,854,775,808to 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

1. byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127.

Uses: It is very useful when working with a stream of data from a network or file.

It is also used when 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;
```

2. short

shortis a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Examples of **short** variable declarations:

```
short s;
short t;
```

3. int

It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.

Use:Variables of type **int**are commonly employed to control loops and to index arrays.

Example: int a

4. long

longis a signed 64-bit type.

Uses: It is used where an **int**type is not large enough to hold the desired value and is useful when big, whole numbers are needed.

Example : long days

long seconds

- **Floating-Point Types**

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.

Example: calculations such as square root. 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. 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

1. float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. 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.

Example:

```
Float hightemp, lowtemp;
```

2. double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

• Characters

In Java, the data type used to store characters is **char**. **char**in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. 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. The range of a **char** is 0 to 65,536.

Program that demonstrates char variables:

```
classCharDemo {
public static void main(String args[]) {
char ch1, ch2;
ch1 = 88; // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

Output:

ch1 and ch2: X Y

• Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**.

Program:

```
classBoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
```

```
// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
```

Output:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Variables

A variable is defined by thecombination of an identifier, a type, and an optional initializer.

Declaring a Variable

All variables must be declared before they can be used.

Syntax *:* *type identifier* [= *value*][, *identifier* [= *value*] ...] ;

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

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

Dynamic Initialization

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

// Program to Demonstrate dynamic initialization.

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

Type Conversion and Casting

Assigning a value of one type to a variable of another type is possible . 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. But 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**.

It is still possible to obtain a conversion between incompatible types. *Casting should be done,* which performs an explicit conversion between incompatible types.

- **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.

Widening conversions works between numeric types. But there are no automatic conversions from the numeric types to **char** or **boolean**.

- **Casting Incompatible Types**

What if we 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 we 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.

General form:

(*target-type*) *value*

**target-type* specifies the desired type to convert the specified value to.

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*.**

Example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.

// Program to 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);
}
}
```

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
```

- **Automatic Type Promotion in Expressions**

  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 sub expression **a * b** is performed using integers—not bytes.

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

- **The Type Promotion Rules**

Java defines several *type promotion* rules that apply to expressions.

1. All **byte**, **short**, and **char** values are promoted to **int**.
2. If one operand is a **long**, the whole expression is promoted to **long**.
3. If one operand is a **float,** the entire expression is promoted to **float**.
4. If any of the operands is **double**, the result is **double**.

Program:

```
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);
}
}
```

- **Arrays**

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its **index.**

- **One-Dimensional Arrays**

A*one-dimensional array* is a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

typevar-name[ ];

Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":

intmonth_days[];

To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days. new is a special operator that allocates memory.

The general form of new as it applies to one-dimensional arrays appears as follows:

array-var = new type[size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array.

month_days = new int[12];

- Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**.

month_days[1] = 28;

The next line displays the value stored at index 3.

System.out.println(month_days[3]);

// Demonstrate a one-dimensional array.

class Array {

```
public static void main(String args[]) {
intmonth_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

It possible to combine the declaration of the array variable with the allocation of thearray itself, as shown here:

```
intmonth_days[] = new int[12];
```

Arrays can be initialized when they are declared.

```
// An improved version of the previous program.
classAutoArray {
public static void main(String args[]) {
intmonth_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}
```

- ## Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a twodimensional array variable called **twoD**.

inttwoD[][] = new int[4][5];

```
// Demonstrate a two-dimensional array.
classTwoDArray {
public static void main(String args[]) {
inttwoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

CHAPTER 2: OPERATORS

Arithmetic Operators

The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	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.

```
// Demonstrate the basic arithmetic operators.
classBasicMath {
public static void main(String args[]) {
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
```

```
double da = 1 + 1;
doubledb = da * 3;
double dc = db / 4;
doubledd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

Output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

- **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.

```
// Demonstrate the % operator.
class Modulus {
```

```
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When you run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

- **Arithmetic Compound Assignment Operators**

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

Syntax: 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.

// Demonstrate several assignment operators.

```
classOpEquals {
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 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++;
```

```
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;
```

```
// Demonstrate ++.
classIncDec {
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);
}
}
```

- **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.

Example:

```
int a = 4;
int b = 1;
```

boolean c = a < b;

In this case, the result of **a<b** (which is **false**) is stored in **c**.

• The Assignment Operator

The *assignment operator* is the single equal sign, =.It has this general form:

var = *expression*;

Here, the type of *var*must be compatible with the type of *expression*

Assignment operator allows 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
```

• The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the **?**.

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.

// 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

Highest

( )	[ ] .		
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		

Lowest

- ### Bitwise Operators

The Bitwise Operators Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte.

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

All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is 00101010, where each position represents a power of two, starting with 20 at the rightmost bit. The next bit position to the left would be 21 , or 2, continuing toward the left with 22 , or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of 21 + 23 + 25 , which is 2 + 8 + 32.

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two's complement, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, –42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42. To decode a negative number, first invert all of the bits, then add 1. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

- ## The Bitwise Logical Operators

The bitwise logical operators are &, |, ^, and ~.

A	B	A | B	A & B	A ^ B	~A
0	0	0	1	0	1
1	0	1	1	1	0

0	1	1	1	1	1
1	1	1	0	0	0

1. The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern: 00101010 becomes 11010101 after the NOT operator is applied.

2. The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
   00101010   42
&  00001111   15
   00001010   10
```

3. The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown

```
   00101010   42
|  00001111   15
   00101111   47
```

4. The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
   00101010   42
^  00001111   15
   00100101   37
```

// Demonstrate the bitwise logical operators.

```
class BitLogic
{
public static void main(String args[])
{
String binary[] = { "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001",
"1010", "1011", "1100", "1101", "1110", "1111" };
int a = 3;          // 0 + 2 + 1 or 0011 in binary
```

```
int b = 6;          // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
 int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

- The Left Shift

The left shift operator, <<< shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << num

Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

```
// Left shifting a byte value.
 class ByteShift
{
public static void main(String args[])
{
byte a = 64, b;
 int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
}
```

```
}
```

The output :

Original value of a: 64

i and b: 256 0

- **The Right Shift**

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is :

value >> num

Here, num specifies the number of positions to right-shift the value in value. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num. T

he following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8:

```
int a = 32;
a = a >> 2;        // a now contains 8
```

00100011 35

>> 2

00001000 8

- **The Unsigned Right Shift**

If you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift. To accomplish this, you will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the >>>. Here, a is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
 a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int

>>>24

00000000 00000000 00000000 11111111 255 in binary as an int

- **Bitwise Operator Compound Assignments**

The following two statements, which shift the value in a right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
class OpBitEquals
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c = 3;
a |= 4;
b >>= 1;
c <<= 1;
 a ^= c;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
 }
}
```

Output:

```
a = 3
b = 1
c = 6
```