1. An Overview of Java

Understanding the basics is very much essential to get the core knowledge of any programming language. This chapter gives the glimpse of programming constructs of Java.

1.1 Object-Oriented Programming

Java is purely object oriented programming (OOP) language. Here, we will discuss the basics of OOPs concepts.

Two Paradigms

Every program consists of two elements viz. code and data. A program is constructed based on two paradigms: *a program written around* ***what is happening*** (known as ***process-oriented model)*** and *a program written around* ***who is being affected*** (known as ***object-oriented model).*** In process oriented model, the program is written as a series of linear (sequential) steps and it is thought of as ***code acting on data.*** Since this model fails to focus on real-world entities, it will create certain problems as the program grows larger.

The object-oriented model focuses on real-world data. Here, the program is organized as data and a set of well-defined interfaces to that data. Hence, it can be thought of as ***data controlling access to code.*** This approach helps to achieve several organizational benefits.

Abstraction

Abstraction can be thought of as ***hiding the implementation details from the end-user***. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows us to layer the semantics of complex systems, breaking them into more manageable pieces. For example, we consider a car as a vehicle and can be thought of as a single object. But, from inside, car is a collection of several subsystems viz. steering, brakes, sound system, engine etc. Again, each of these subsystems is a collection of individual parts (Ex. Sound system is a combination of a radio and CD/tape player). As an owner of the car, we manage it as an individual entity by achieving hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to *do something.* This is the essence of object-oriented programming.

OOPs Principles

Encapsulation, Inheritance and Polymorphism are the basic principles of any object oriented programming language.

Encapsulation is the mechanism to bind the data and code working on that data into a single entity. It provides the security for the data by avoiding outside manipulations. In Java, encapsulation is achieved using ***classes.*** A ***class*** is a collection of data and code. An ***object*** is an ***instance*** of a class. That is, several objects share a common structure (data) and behavior (code) defined by that class. A class is a logical entity (or prototype) and an object is a physical entity. The elements inside the class are known as ***members***. Specifically, the data or variables inside the class are called as ***member variables*** or ***instance variables*** or ***data members.*** The code that operates on these data is referred to as ***member***

methods or ***methods*** (In C++, we term this as member function). The method operating on data will define the behavior and interface of a class.

Another purpose of the class is to hide the information from outside manipulation. Class uses ***public*** and ***private*** interfaces. The members declared as private can only be accessed by the members of that class, whereas, the public members can be accessed from outside the class.

Inheritance allows us to have code re-usability. It is a process by which one object can acquire the properties of another object. It supports the concept of hierarchical classification. For example, consider a large group of animals having few of the abstract attributes like size, intelligence, skeletal structure etc. and having behavioral aspects like eating, breathing etc. Mammals have all the properties of Animals and also have their own specific features like type of teeth, mammary glands etc. that make them different from Reptiles. Similarly, Cats and Dogs have all the characteristics of mammals, yet with few features which are unique for themselves. Though Doberman, German-shepherd, Labrador etc. have the features of Dog class, they have their own unique individuality. This concept can be depicted using following figure.

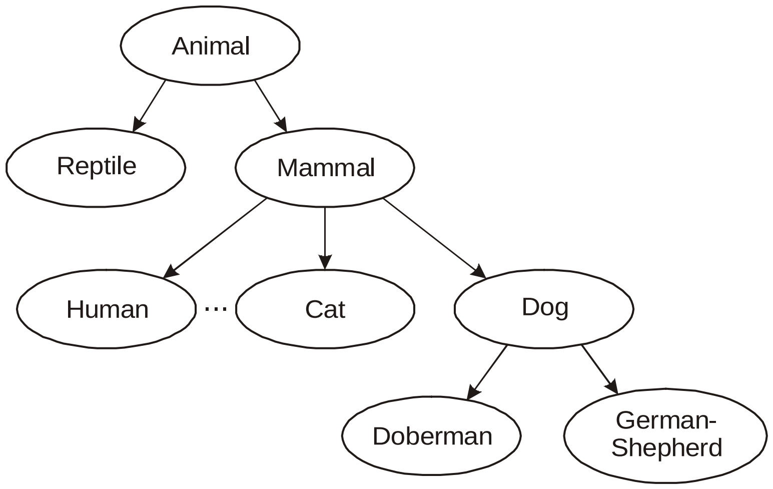

Figure 1.1 Example of Inheritance

If we apply the above concept for programming, it can be easily understood that a code written is reusable. Thus, in this mechanism, it is possible for one object to be a specific instance of a more general case. Using inheritance, an object need only define those qualities that make it a unique object within its class. It can inherit its general attributes from its parent. Hence, through inheritance, we can achieve *generalization-specialization* concept. The top-most parent (or base class or ***super class***) class is the generalized class and the bottom-most child (or derived class or ***subclass***) class is a more specialized class with specific characteristics.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization. This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

Polymorphism can be thought of as *one interface, multiple methods.* It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider an example of performing stack operation on three different types of data viz. integer, floating-point and characters. In a non-object oriented programming, we write functions with different names for push and pop operations though the logic is same for all the data types. But in Java, the same function names can be used with data types of the parameters being different.

NOTE: To understand the further topics of this chapter, we should know few basic concepts.

The key features of Java are ***security*** and ***portability (platform-independent nature)***. When we download any application from the internet, there is a chance that the downloaded code contain virus. But, downloading the Java code assures security. Java program can run on any type of system connected to internet and thus provides portability.

The Platform independent nature can be interpreted by two things:
Operating System Independent: Independent of the operating system on which your source code is being run.
Hardware Independent: Doesn't depend upon the hardware on which your java code is run upon i.e. it can run on any hardware configuration.

These two points make it a platform independent language. Hence, the users do not have to change the syntax of the program according to the Operating System and do not have to compile the program again and again on different Operating Systems. The meaning of this point can be understood as you read further.

Now talking about C,C++ they are platform dependent languages as the file which compiler of C,C++ forms is a **.exe**(executable) file which is operating system dependent. The C++ program is controlled by the operating system whereas, the execution of a Java program is controlled by ***JVM (Java Virtual Machine).***

The JVM is the Java run-time system and is the main component of making the java a platform independent language. For building and running a java application we need JDK(Java Development Kit) which comes bundled with ***Java runtime environment(JRE)*** and JVM. With the help of JDK the user compiles and runs his java program. As the compilation of java program starts the ***Java Bytecode*** is created i.e. a **.class** file is created by JRE. Bytecode is a highly optimized set of instructions designed to be executed by JVM. Now the JVM comes into play, which is made to read and execute this bytecode. The JVM is linked with operating system and runs the bytecode to execute the code depending upon operating system. Therefore, a user can take this class file(Bytecode file) formed to any operating system which is having a JVM installed and can run his program easily without even touching the syntax of a program and without actually having the source code. The .class file which consists of bytecode is not user-understandable and can be interpreted by JVM only to build it into the machine code.

Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs. Java also has the standard data size irrespective of operating system or the processor. These features make the java as a portable (platform-independent) language.

Usually, 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. To improve the performance, Java provides a ***Just-in-time (JIT)*** compiler for bytecode. JIT compilers alter the role of the JVM a little by directly compiling Java bytecode into native platform code, thereby relieving the JVM of its need to manually call underlying native system services. When JIT compiler is installed, instead of the JVM calling the underlying native operating system, it calls the JIT compiler. The JIT compiler in turn generates native code that can be passed on to the native operating system for execution. This makes the java program to run faster than expected.

Moreover, 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. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted.

1.2 A First Simple Program

Here, we will discuss the working of a Java program by taking an example –

Program 1.1

```
class Prg1
{
        public static void main(String args[ ])
        {
                System.out.println("Hello World!!!");
        }

}
```

Save this program as ***Prg1.java***. A java program source code is a text file containing one or more class definitions is called as ***compilation unit*** and the extension of this file name should be ***.java.***

To compile above program, use the following statement in the command prompt –

```
javac Prg1.java
```

(**Note:** You have to store the file Test.java in the same location as that of *javac* compiler or you should set the Environment PATH variable suitably.)

Now, the *javac* compiler creates a file **Prg1*.class*** containing bytecode version of the program, which can be understandable by JVM. To run the program, we have to use Java application launcher called ***java***. That is, use the command –

```
Java Prg1
```

The output of the program will now be displayed as –

```
Hello World!!!
```

Note: When java source code is compiled, each class in that file will be put into separate output file having the same name as of the respective class and with the extension of **.class**. To run a java code, we need a class file containing *main()* function (Though, we can write java program without *main()*, for the time-being you assume that we need a *main()* function!!!). Hence, it is a tradition to give the name of the java source code file as the name of the class containing *main()* function.

Let us have closer look at the terminologies used in the above program now –

class	is the keyword to declare a class.
Test	is the name of the class. You can use any valid identifier for a class name.
main()	is name of the method from which the program execution starts.
public	is a keyword indicating the access specifier of the method. The *public* members can be accessed from outside the class in which they have been declared. The *main()* function must be declared as *public* as it needs to be called from outside the class.
static	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.
void	indicates that *main()* method is not returning anything.
String args[ ]	The *main()* method takes an array of *String* objects as a command-line argument.
System	is a predefined class (present in java.lang package) which gives access to the system. It contains pre-defined methods and fields, which provides facilities like standard input, output, etc.
out	is a static final (means not inheritable) field (ie, variable)in System class which is of the type PrintStream (a built-in class, contains methods to print the different data values). Static fields and methods must be accessed by using the class name, so we need to use *System.out*.
println	is a public method in PrintStream class to print the data values. After printing the data, the cursor will be pushed to the next line (or we can say that, the data is followed by a *new line*).

1.3 A Second Short Program

Here, we will discuss a program having variables. Variable is a named memory location which may be assigned a value in the program. A variable can be declared in the java program as –

type var_name;

Here, ***type*** is any built-in or user-defined data type (We will discuss various data types later in detail). ***var_name*** is any valid name given to the variable. Consider the following example –

Program 1.2

```
class Prg2
{
        public static void main(String args[])
        {
                int n;
                n=25;
                System.out.println(“The value of n is: “ + n);
                n= n*3;
                System.out.print(“The current value of n is: “ );
                System.out.println(n);
        }
}
```

The output will be –

```
The value of n is: 25
The current value of n is: 75
```

In the above program, we have declared an *integer* variable ***n*** and then assigned a value to it. Now, observe the statement,

```
System.out.println("The value of n is: " + n);
```

Here, we are trying to print a string value *"The value of n is:"* and also value of an integer *n* together. For this, we use + symbol. Truly speaking, the value of *n* is internally converted into *string* type and then concatenated with the string *"The value of n is:"*. We can use + symbol as many times as we want to print several values.

The above program uses one more method *System.out.print()* which will keep the cursor on the same line after displaying the output. That is, no *new line* is not included in it.

1.4 Using Blocks of Code

Java allows two or more statements to be grouped into *blocks of code,* also called *code blocks.* This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.

For example, a block can be a target for Java's **if** and **for** statements. Consider this **if** statement:

```
if(x < y)
{  // block begins
        x = y;
        y = 0;
} // block ends here
```

The main reason for the existence of blocks of code is to create logically inseparable units of code.

1.5 Lexical Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. We will discuss the significance of each of these here.

Whitespace : In Java, whitespace is a space, tab or newline. Usually, a space is used to separate tokens; tab and newline are used for indentation.

Identifiers : Identifiers are used for class names, method names, and variable names. An identifier may be any sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number. As Java is case-sensitive, **Avg** is a different identifier than **avg**.
Examples of valid identifiers: Avg, sum1, $x, sum_sq etc.
Examples of invalid identifiers: 2sum, sum-sq, x/y etc.

Literals : A constant value in Java is created by using a *literal* representation of it. For example, 25 (an integer literal), 4.5 (a floating point value), 'p' (a character constant, "Hello World" (a string value).

Comments : There are three types of comments defined by Java. Two of these are well-know viz.

Single-line comment(starting with //), ***multiline comment*** (enclosed withing /* and */).
The third type of comment viz. ***documentation comment*** is used to produce an HTML file that documents your program. The documentation comment begins with a **/**** and ends

with a ***/**.

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

Symbol	Name	Purpose
( )	Parentheses	Used to provide parameter list in method definition and to call methods. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to initialize arrays, to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types, to dereference 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 sub-packages and classes. Also used to separate a variable or method from a reference variable.

Keywords : There are 50 keywords currently defined in the Java language as shown in the following table. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method.

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

The keywords **const** and **goto** are reserved but are rarely used. 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.

1.6 The Java Class Libraries

The sample programs discussed in previous sections make use of two of Java's built-in methods: **println()** and **print( )**. As mentioned, these methods are members of the **System** class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java is a combination of the Java language itself, plus its standard classes. The class libraries provide much of the functionality that comes with Java. The standard library classes and methods are described in detail in forthcoming chapters.

1.7 Java is a Strongly Typed Language

A strongly-typed programming language is one in which each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types. Certain operations may be allowable only with certain data types.

In other words, every variable has a type, every expression has a type, and every type is strictly defined. And, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class. These features of Java make it a strongly typed language.

1.8 The Primitive Types

Java defines eight ***primitive*** (or ***simple***) data types viz.

- ***byte, short, int, long*** : belonging to *Integers* group involving whole-valued signed numbers.
- ***char*** : belonging to *Character* group representing symbols in character set like alphabets, digits, special characters etc.
- ***float, double*** : belonging to *Floating-point* group involving numbers with fractional part.
- ***boolean*** : belonging to *Boolean* group, a special way to represent true/false values.

These types can be used as primitive types, derived types (arrays) and as member of user-defined types (classes). All these types have specific range of values irrespective of the platform in which the program being run. In C and C++ the size of integer may vary (2 bytes or 4 bytes) based on the platform. Because of platform-independent nature of Java, such variation in size of data types is not found in Java, and thus making a Java program to perform better.

Integers

Java defines four integer types viz. ***byte, short, int*** and ***long.*** All these are signed numbers and Java does not support unsigned numbers. 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 (in bits)**	**Range**
long	64	-2^{64} to $+2^{64}-1$
int	32	-2^{32} to $+2^{32}-1$
short	16	-2^{16} to $+2^{16}-1$ (-32768 to +32767)
byte	8	-2^{8} to $+2^{8}-1$ (-128 to +127)

Each of the integer types are discussed below.

byte **:** This is the smallest integer type. Variables of type **byte** are especially useful when you are working with a stream of data from a network or file. They are also useful when you are 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,

```
byte b, c;
```

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

long **:** It 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.

Following is an example showing need for *long* variables:

Program 1.3

```
class Light
{
        public static void main(String args[ ])
        {
                int lightspeed;
                long days, seconds, distance;

                // approximate speed of light in miles per second
                lightspeed = 186000;
                days = 1000;                        // specify number of days here

                seconds = days * 24 * 60 * 60;      // convert to seconds
                distance = lightspeed * seconds;    // compute distance

                System.out.print("In " + days);
                System.out.print(" days light will travel about ");
                System.out.println(distance + " miles.");
        }
}
```

The output will be –

```
In 1000 days light will travel about 16070400000000 miles.
```

Floating –Point Types

Floating-point (or real) numbers are used when evaluating expressions that require fractional precision. 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)	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 currencies, temperature etc. Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

double **:** 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.

Here is a short program that uses **double** variables to compute the area of a circle:

Program 1.4

```
class Area
{
        public static void main(String args[])
        {
                double pi, r, a;
                r = 10.8;
                pi = 3.1416;
                a = pi * r * r;
                System.out.println("Area of circle is " + a);
        }
}
```

The output would be –

```
Area of circle is 366.436224
```

Characters

In Java, **char** is the data type used to store characters. In C or C++, ***char*** is of 8 bits, whereas in Java it requires 16 bits. Java uses Unicode to represent characters. **Unicode** is a computing industry standard for the consistent encoding, representation and handling of text expressed in many languages of the world. Unicode has a collection of more than 109,000 characters covering 93 different languages like Latin, Greek, Arabic, Hebrew etc. That is why, it requires 16 bits. The range of a **char** is 0 to 65,536. 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. Though it seems to be wastage of memory as the languages like English, German etc. can accommodate their character set in 8 bits, for a global usage point of view, 16-bits are necessary.

Though, ***char*** is designed to store Unicode characters, we can perform arithmetic operations on them. For example, we can add two characters (but, not char variables!!), increment/decrement character variable etc. Consider the following example for the demonstration of characters.

Program 1.5

```
class CharDemo
{
        public static void main(String args[])
        {
                char ch1=88, ch2='Y';

                System.out.print("ch1 and ch2: ");
                System.out.println(ch1 + " " + ch2);

                 ch1++;          //increment in ASCII (even Unicode) value
                System.out.println("ch1 now contains "+ch1);

                --ch2;           //decrement in ASCII (even Unicode) value
                System.out.println("ch2 now contains "+ch2);

                /*   ch1=35;
                     ch2=30;
                     char ch3;

                      ch3=ch1+ch2;   //Error
                  */

                ch2='6'+'A';   //valid
                System.out.println("ch2 now contains "+ch2);
        }
}
```

The output would be –

```
ch1 and ch2: X Y
ch1 now contains Y
ch2 now contains X
ch2 now contains w
```

Booleans

For storing logical values (***true*** and ***false***), Java provides this primitive data type. Boolean is the output of any expression involving relational operators. For control structures (like if, for, while etc.) we need to give boolean type. In C or C++, false and true values are indicated by zero and a non-zero numbers respectively. And the output of relational operators will be 0 or 1. But, in Java, this is not the case. Consider the following program as an illustration.

Program 1.6

```
class BoolDemo
{
        public static void main(String args[])
        {
                boolean b;
                b = false;

                System.out.println("b is " + b);
                b = true;
                System.out.println("b is " + b);

                if(b)
                        System.out.println("True block");

                b = false;
                if(b)
                        System.out.println("False Block will not be executed");

                b=(3<5);
                System.out.println("3<5 is " +b);
        }
}
```

The output would be –

```
b is false
b is true
True block
3<5 is true
```

NOTE: Size of a Boolean data type is JVM dependent. But, when Boolean variable appears in an expression, Java uses 32-bit space (as int) for Boolean to evaluate expression.

1.9 A Closer Look at Literals

A *literal* is the source code representation of a fixed value. In other words, by literal we mean any number, text, or other information that represents a value. Literals are represented directly in our code without requiring computation. Here we will discuss Java literals in detail.

Integer Literals

Integers are the most commonly used type in the typical program. Any whole number value is an integer literal. For example, 1, 25, 33 etc. These are all decimal values, having a base 10. With integer literals we can use *octal* (base 8) and *hexadecimal* (base 16) also. Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, a value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. Hexadecimal constants denoted with a leading zero-x, (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f)* are substituted for 10 through 15.

Integer literals create an **int** value, which in Java is a 32-bit integer value. It is possible to assign an integer literal to other integer types like byte or long. When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase ***L*** to the literal. For example, 0x7fffffffffffffffL or 9223372036854775807L is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

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 less-accurate **float** type requires only 32 bits.

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, they 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 new-line 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. Following table shows the character escape sequences.

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

String Literals
String literals are a sequence of characters enclosed within a pair of double quotes. Examples of string literals are

“Hello World” “two\nlines” “\”This is in quotes\”“

Java strings must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages. In Java, strings are actually objects and are discussed later in detail.

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

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

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

The *type* is any of primitive data type or class or interface. The *identifier* is the name of the variable. We can initialize the variable at the time of variable declaration. 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=5, c;         // declares three ints, a, b, and c where b is initialized to 5
byte z = 22;
double pi = 3.1416;
char x = '$';
```

Dynamic Initialization
Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example,

```
int a=5, b=4;
int c=a*2+b;    //variable declaration and dynamic initialization.
```

The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables
A variable in Java can be declared within a block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope which determines the accessibility of variables and/or objects defined within it. It also determines the lifetime of those objects.

Many languages like C/C++ have two scopes viz. global and local. But in Java, every line of code should be embedded within a class. That is, no code is written outside the class. So, usage of the terms *global* and *local* makes no sense. Instead, Java has two scopes viz. ***class level scope*** and ***method (or function) level scope***. Class level scope is discussed later and we will discuss method scope here.

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. 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.

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.

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

Program 1.7
```
class Scope
{
        public static void main(String args[])
        {
                int x=10, i;        // x is local to main()

                if(x == 10)
                {
                        int y = 20;        // y is local to this block
                        System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                }

                // inclusion of following line generates Error
                // y = 100;

                System.out.println("x is " + x);

                for(i=0;i<3;i++)
                {
                        int a=3;
                        System.out.println("a is " + a);
                        a++;
                }
        }
}
```
The output would be –
```
x and y: 10 20
x is 40
a is 3
a is 3
a is 3
```

Note that, variable ***a*** is declared within the scope of ***for*** loop. Hence, each time the loop gets executed, variable ***a*** is created newly and there is no effect of *a*++ for next iteration.

NOTE:
In Java, same variable name can not be used in nested scopes. That is, the following code snippet generates error.

```
class Test
{
        public static void main(String args[])
        {
                int x=3;

                {
                        int x=5;        //error!!
                }
        }
}
```

(**Note that, having same variable name in nested scopes is VALID in C/C++).**

1.11 Type Conversion and Casting

It is quite common in a program to assign value of one type to a variable of another type. If two types are compatible, Java performs ***implicit type conversion***. For example, *int* to *long* is always possible. But, whenever the types at two sides of an assignment operator are not compatible, then Java will not do the conversion implicitly. For that, we need to go for ***explicit type conversion*** or ***type casting.***

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**.

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, we 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,

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

When a floating-point value is assigned to an integer type, the fractional component is lost. And such conversion is called as ***truncation.*** 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. Following program illustrates various situations of explicit casting.

Program 1.8

```
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);
        }
}
```

The output would be –

```
Conversion of int to byte:      i = 257           b = 1
Conversion of double to int:    d = 323.142       i = 323
Conversion of double to byte:   d = 323.142       b = 67
```

Here, 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.

1.12 Automatic Type promotion in Expression

Apart from assignments, type conversion may happen in expressions also. In an arithmetic expression involving more than one operator, some intermediate operation may exceed the size of either of the operands. For example,

```
byte x=25, y=80, z=50;
int p= x*y/z ;
```

Here, the result of operation x*y is 4000 and it exceeds the range of both the operands i.e. byte (-128 to +127). In such a situation, Java promotes byte, short and char operands to int That is, the operation x*y is performed using int but not byte and hence, the result 4000 is valid.

On the other hand, the automatic type conversions may cause error. For example,

```
byte x=10;
byte y= x *3;   //causes error!!!
```

Here, the result of x *3 is 30, and is well within the range of byte. But, for performing this operation, the operands are automatically converted to byte and the value 30 is treated as of int type. Thus, assigning an int to byte is not possible, which generates an error. To avoid such problems, we should use type casting. That is,

```
byte x=10;
byte y=(byte) (x *3);   //results 30
```

Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows:

- All **byte**, **short**, and **char** values are promoted to **int.**
- 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 is **double**, the result is **double**.

Program 1.9

```
class TypePromo
{
    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("result = " + result);
    }
}
```

The output would be –

```
result = 626.7784146484375
```

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.

1.13 Arrays

Array is a collection of related items of same data type. Many items of an array share common name and are accessed using index. Array can be one dimensional or multi-dimensional.

One Dimensional Arrays

It is a list of related items. To create 1-d array, it should be declared as –

type arr_name[];

Here, ***type*** determines the data type of elements of ***arr_name***. In Java, the above declaration will not allocate any memory. That is, there is no physical existence for the array now. To allocate memory, we should use ***new*** operator as follows:

arr_name=new type[size];

Here, **size** indicates number of elements in an array. The **new** keyword is used because, in Java array requires dynamic memory allocation. The above two statements can be merged as –

type arr_name[]=new type[size];

For example, following statement create an array of 10 integers –

```
int arr[ ]=new int[10];
```

Array index starts with 0 and we can assign values to array elements as –

```
arr[0]=25;
arr[1]=32; and so on.
```

Arrays can be initialized at the time of declaration. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example –

```
int arr[ ] ={1, 2, 3, 4};
```

The above statement creates an integer array of 4 elements.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will get a run-time error.

Multidimensional Arrays

Multidimensional arrays are arrays of arrays. Here, we will discuss two dimensional arrays in Java. The declaration of 2-d array is as follows –

type arr_name[][]=new type[row_size][col_size];

here, ***row_size*** and ***col_size*** indicates number of rows and columns of 2-d arrays. In other words, ***row-size*** indicates number of 1-d arrays and ***col_size*** indicates size of each of such 1-d array. Consider the following program –

Program 1.10

```
class TwoDArray
{
	public static void main(String args[])
	{
		int twoD[][]= new int[3][4];
		int i, j;

		for(i=0; i<3; i++)
			for(j=0; j<4; j++)
				twoD[i][j] = i+j;

		for(i=0; i<3; i++)
		{
			for(j=0; j<4; j++)
				System.out.print(twoD[i][j] + " ");

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

The output would be –

```
0 1 2 3
1 2 3 4
2 3 4 5
```

Instead of allocating memory for 2-day as shown in the above program, we can even do it in a different way. We can first mention ***row_size*** and then using different statements, mention ***col_size*** as shown below –

```
int twoD[][]= new int[3][];
twoD[0]=new int[4] ;
twoD[1]=new int[4] ;
twoD[2]=new int[4] ;
```

But, above type of allocation is not having any advantage unless we need ***uneven or irregular*** multidimensional array. In Java, it is possible to have different number of columns for each row in a 2-d array. For example,

Program 1.11

```
class UnevenArr
{
	public static void main(String args[])
	{
```

```
                int twoD[][] = new int[3][];
                twoD[0] = new int[3];
                twoD[1] = new int[1];
                twoD[2] = new int[5];

                int i, j, k = 0;

                for(i=0; i<3; i++)
                        for(j=0; j<twoD[i].length; j++, k++)
                                twoD[i][j] = k;

                for(i=0; i<3; i++)
                {
                        for(j=0; j<twoD[i].length; j++)
                                System.out.print(twoD[i][j] + " ");
                        System.out.println();
                }
        }
}
```

The output would be –

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

Here, we have declared a 2-d array with 3 rows. But, number of columns for each row varies. The first 1-d array has 3 elements, second 1-d array as a single element and the third 1-d array has 5 elements.

A 2-d array can be initialized at the time of declaration as follows –

```
int a[ ][ ]={{1,2},{3,4} };
```

We can have more than 2 dimensions as –

```
int a[ ][ ][ ]=new int[3][2][4];
```

Here, the array elements can be accessed using 3 indices like *a[i][j][k]*.

Alternative Array Declaration Syntax

There is another way of array declaration as given below –

type[] arr_name;

That is, following two declarations are same –

```
int a[ ]=new int[3];
int[ ] a= new int[3];
```

Both the declarations will create an integer array of 3 elements. Such declarations are useful when we have multiple array declarations of same type. For example,

```
int [ ] a, b, c;
```

will declare three arrays viz. a, b and c of type integer. This declaration is same as –

```
int a[ ], b[ ], c[ ];
```

The alternative declaration form is also useful when specifying an array as a return type for a method.

1.14 A few words about Strings

In Java, String is a class but not array of characters. So, the features of String class can be better understood after learning about the concepts of classes in further chapters. For the time-being, we will glance at **String** type.

- We can have array of strings.
- A set of characters enclosed within double quotes can be assigned to a String variable.
- One variable of type String can be assigned another String variable.
- Object of type String can be used as an argument to **println** as –

```
String str="Hello";
System.out.println(str);
```

1.15 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	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

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 a subset of **int**. Note down following few points about various operators.

- Basic arithmetic operators like +, -, * and / behave as expected for numeric data.
- The – symbol can be used as unary operator to negate a variable.
- If / is operated on two integer operands, then we will get only integral part of the result by truncating the fractional part.
- The % operator returns the remainder after division. It can be applied on integer and floating-point types. For example,

```
int x=57;
double y= 32.8;
System.out.println("on integer " + x%10);     //prints 7
System.out.println("on double " + y%10);      //prints 2.8
```

- Compound assignment operators like += will perform arithmetic operation with assignment. That is,

```
a+=2;   implies      a=a+2;
```

- Increment/decrement operators (++ and --) will increase/decrease the operand by 1. That is,

```
a++;    implies      a=a+1;
b--;    implies      b=b-1;
```

- The ++ and -- operators can be used either as pre-increment/decrement or post-increment/decrement operator. For example,
```
x= 5;
y=x++;
```
Now, y will be 5 and x is then incremented to be 6.
```
x= 5;
y=++x;
```
Now, x is incremented to become 6 and then 6 is assigned to y.

NOTE that in C/C++, the % operator can not be used on float or double and should be used only on integer variable.

1.16 Bitwise Operators

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

Operator	**Meaning**
~	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

Since bitwise operators manipulate the bits within the integer, let us first understand the bit-representation of integer data in Java.

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 2^0 at the rightmost bit. All of the integer types are signed integers. 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, and then add 1. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

Bitwise Logical Operators

The bitwise logical operators are &, |, ^ and ~. Following table shows the result of each operation.

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

Bitwise NOT

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

Bitwise AND

As the name suggests, initially, operands are converted into binary-format. Then, the AND (&) operation is performed on the corresponding bits of operands. Consider an example –

```
int x=5, y=6,z;
z= x & y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0101
y ⇨   &   0000 0110
          ---------
z ⇨       0000 0100
```

Thus, z will be decimal equivalent of 0000 0100, which is 4.

Bitwise OR

Here, the OR (|) operations is performed on individual bit of operands. For example –

```
int x=5, y=6,z;
z= x | y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0101
y ⇨   |   0000 0110
          ---------
z ⇨       0000 0111
```

Thus, z will be decimal equivalent of 0000 0111, which is 7.

Bitwise XOR

In XOR operation, if both bits are same (either both are 1 or both 0), then the resulting bit will be 0 (false). Otherwise, the resulting bit is 1 (true). For example –

```
int x=5, y=6,z;
z= x ^ y;
```

Now, this operation is carried out as –

$$\begin{array}{lcll} x & \Rightarrow & & 0000\ 0101 \\ y & \Rightarrow & \wedge & \underline{0000\ 0110} \\ z & \Rightarrow & & 0000\ 0011 \end{array}$$

Thus, z will be decimal equivalent of 0000 0011, which is 3.

Left Shift

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

value << num

For each shift, one higher order bit is shifted out (or lost) and extra zero is appended as the lower order bit. Thus, for int, after 31 shifts, all the bits will be lost and result will be 0, whereas for long, after 63 shifts, all bits will be lost.

Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values. As you know, **byte** and **short** values are promoted to **int** when an expression is evaluated. Furthermore, the result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shifted for 31 times. To avoid this problem, we should use type-casting as shown in the following example.

Program 1.12

```
class ShiftDemo
{
	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 result would be –

```
Original value of a: 64
i and b: 256    0
```

Since **a** is promoted to **int** for evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero.

Each left shift can be thought of as multiplying the number by 2. But, one should be careful because once the number crosses its range during left shift, it will become negative. Consider an illustration –

Program 1.13

```
class ShiftDemo1
 {
	public static void main(String args[])
```

```
                {
                        int i;
                        int num = 0xFFFFFFE;
                        for(i=0; i<4; i++)
                        {
                                num = num << 1;
                                System.out.println(num);
                        }
                }
        }
```

The output would be –

```
        536870908
        1073741816   //twice the previous value
        2147483632   //twice the previous value
        -32          //crosses the range of int and hence negative
```

Right Shift

The right shift operator, **>>** shifts all of the bits in a value to the right by a specified number of times. It has this general form:

value >> num

For each shift, one lower order bit is shifted out (or lost) and extra zero is appended as the higher order bit. For example,

```
        int a = 35;        //00100011 is the binary equivalent
        a = a >> 2;        // now, a contains 8
```

Each right shift can be thought of as dividing the number by 2. When you are shifting right, the top (leftmost) bit is filled with the previous content of the top bit. This is called ***sign extension*** and is needed to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

```
                11111000    (–8)
                >>1
                11111100    (–4)
```

Unsigned Right Shift

We have seen that right shift always fills the highest order bit with the previous content of the top bit. But when we are using shift operation on non-numeric data, sign-bit has no significance. To ignore the sign-bit, we will go for unsigned right shift. 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 Assignment

We can use compound assignment even with bitwise operators. That is,

```
a<<=2;       implies a=a<<2;
a^=3;        implies a=a^3;       and so on.
```

1.17 Relational Operators

The relational operators determine the relationship between two operands. Specifically, they determine equality and ordering among operands. Following table lists the relational operators supported by Java.

Operator	Meaning
==	Equal to (or comparison)
!=	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. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test, **!=**. Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other. For example, the following code fragment is perfectly valid:

```
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**.

Note that in C/C++ we can have following type of statement –

```
int flag;
…….
if(flag)
   //do something
```

In C/C++, ***true*** is any non-zero number and ***false*** is zero. But in Java, ***true*** and ***false*** are Boolean values and nothing to do with zero or non-zero. Hence, the above set of statements **will cause an error** in Java. We should write –

```
int flag;
………..
if(flag==1)
   //do some thing
```

1.18 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	Meaning
&	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 truth table is given below for few operations:

A	**B**	**A|B**	**A&B**	**A^B**	**!A**
False	False	False	False	False	True
False	True	True	False	True	True
True	False	True	False	True	False
True	True	True	True	False	False

Consider a program which illustrates above operations:

Program 1.14

```
class BoolLogic
{
        public static void main(String args[])
        {
                boolean a = true;
                boolean b = false;
                boolean c = a | b;
                boolean d = a & b;
                boolean e = a ^ b;
                boolean f = (!a & b) | (a & !b);
                boolean g = !a;

                System.out.println(" a = " + a);
                System.out.println(" b = " + b);
                System.out.println(" a|b = " + c);
                System.out.println(" a&b = " + d);
                System.out.println(" a^b = " + e);
                System.out.println("!a&b|a&!b = " + f);
                System.out.println(" !a = " + g);

                boolean h = b & (a=!a);

                System.out.println("b & (a=!a) =" +h);
                System.out.println("New a is "+a);
        }
}
```

The output would be –

```
a = true
b = false
```

```
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
b & (a=!a) =false
New a is false
```

Note: In C/C++, the logical AND/OR operations never evaluates the second operand if the value of first operand itself can judge the result. That is, if the first operand is *false*, then second operand is not evaluated in AND operation and result will be false. Similarly, if the first operand is *true* in OR operation, without evaluating the second operand, it results *true.* But in Java, Boolean logical operators will not act so. Even if the first operand is decisive, the second operand is evaluated. This can be observed in the above program while evaluating h= b& (a= !a). Here, *b* is *false* and hence ANDed with anything results *false*. But, still the second operand (a= !a) is evaluated resulting *a* as false.

If we don't want the second operand to be evaluated, we can use *short-circuit logical* operators.

Short-Circuit Logical Operators

The short-circuit AND (&&) and OR (||) operators will not evaluate the second operand if the first is decisive. For example,

```
int x=0, n=5;
……..
if(x!=0 && n/x > 0)
        //do something
```

Here, the first operand x!= 0 is false. If we use logical AND (&) then the second operand n/x>0 will be evaluated and we will get DivisionByZero Exception. So, to avoid this problem we use && operator which will never evaluated second operand if the first operand results into false.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100)
        d = 100;
```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

1.19 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.* 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.

1.20 The ?: Operator

Java supports *ternary operator* which sometimes can be used as an alternative for *if*-then-*else* statement. The general form is –

var = expression1 **?** *expression2* **:** *expression3;*

Here, *expression1* is evaluated first and it must return Boolean type. If it results *true*, then value of *expression2* is assigned to *var*, otherwise value of *expression3* is assigned to *var.* For example,

```
int a, b, c ;
……….
c= (a>b)?a:b;          //c will be assigned with biggest among a and b
```

1.21 Operator Precedence

Following table describes the precedence of operators. Though parenthesis, square brackets etc. are separators, they do behave like operators in expressions. Operators at same precedence level will be evaluated from left to right, whichever comes first.

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

1.22 Using Parentheses

Parentheses always make the expression within them to execute first. This is necessary sometimes. For example,

```
a= b – c * d;
```

Here, c and d are multiplied first and then the result is subtracted from b. If we want subtraction first, we should use parenthesis like

```
a= (b-c)*d;
```

Sometimes, parenthesis is useful for clarifying the meaning of an expression and for making readers to understand the code. For example,

```
a | 4 + c >> b & 7       can be written as –
(a | (((4 + c) >> b) & 7))
```

In such situations, though parenthesis seems to be redundant, it existence will not reduce the performance of the program.

1.23 Control Statements

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.

1.24 Java's Selection

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 Statement

The general form is –

```
if (condition)
{
        //true block
}
else
{
        //false block
}
```

If the *condition* is true, then the statements written within *true block* will be executed, otherwise *false block* will be executed. The *condition* should result into **Boolean** type. For example,

```
int a, b, max;
…………
if(a>b)
        max=a;
else
        max=b;
```

Nested-if Statement

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. For 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)
```

The if-else-if Statement

The general form is –

```
if(condition1)
        block1;
else if(condition2)
        block2;
…………..
…………..
else
        blockn
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the block associated with that **if** is executed, and the rest of the ladder is bypassed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.

switch Statement

The **switch** statement is Java's multi-way 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
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken. The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**. The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.

NOTE:

- We can even nest switch statements one within the other.
- 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 **if**s.

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a **switch** statement, the Java compiler will inspect each of the **case** constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

1.25 Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.* A loop repeatedly executes the same set of instructions until a termination condition is met.

while Loop

The general form is –

```
while(condition)
{
        //body of the 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.

do- while Loop

The general form is –

```
do
{
        //body of the 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.

for Loop

The general form is –

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

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 only executed 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.

for-each Loop

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. 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 base type of the array.

Consider an example –

```
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];
```

The above set of statements can be optimized as follows –

```
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.

For multi-dimensional arrays:

The for-each version also works for multi-dimensional arrays. Since a 2-d array is an array of 1-d array, the iteration variable must be a reference to 1-d array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions.

Consider the following example –

Program 1.15

```
class ForEach
{
	public static void main(String args[])
	{
		int sum = 0;
		int nums[][] = new int[2][3];

		// give nums some values
		for(int i = 0; i < 2; i++)
			for(int j=0; j < 3; j++)
				nums[i][j] = (i+1)*(j+1);

		for(int x[ ] : nums)	//nums is a 2-d array and x is 1-d array
		{
			for(int y : x)	// y refers elements in 1-d array x
			{
				System.out.println("Value is: " +y);
				sum += y;
			}
		}
		System.out.println("Summation: " + sum);
	}
}
```

The output would be –

```
Value is: 1
Value is: 2
Value is: 3
Value is: 2
Value is: 4
Value is: 6
Summation: 18
```

The for-each version of **for** has several applications viz. Finding average of numbers, finding minimum and maximum of a set, checking for duplicate entry in an array, searching for an element in unsorted list etc. The following program illustrates the sequential (linear) search.

Program 1.16

```
class SeqSearch
{
	public static void main(String args[])
	{
		int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
		int val = 5;
		boolean found = false;

		for(int x : nums)
		{
			if(x == val)
			{
```

```
                        found = true;
                        break;
                    }
                }
                if(found)
                    System.out.println("Value found!");
            }
        }
```

The output would be –

```
        Value found !
```

1.26 Jump Statements

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

Using break

In java, ***break*** can be used in 3 different situations:

- To terminate statement sequence in **switch**
- To exit from a loop
- Can be used as a *civilized* version of ***goto***

Following is an example showing terminating a loop using break.

```
    for (int i=0;i<20;i++)
        if(i==5)
            break;
        else
            System.out.println(“ i= “ + i);
```

The above code snippet prints values from 0 to 4 and when i become 5, the loop is terminated.

Using break as a form of goto

Java does not have a goto statement because it is an un-conditional jump and may end up with an infinite loop. But in some situations, goto will be useful. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems. The general form of labeled break is:

break label;

Following program illustrates the usage of labeled break.

Program 1.17

```
    class Break
    {
        public static void main(String args[])
        {
            boolean t = true;
```

```
                    first: {
                            second: {
                                    third: {
                                            System.out.println("Before the break.");
                                            if(t)
                                              break second;          // break out of second block
                                            System.out.println("This won't execute");
                                          }
                                        System.out.println("This won't execute");
                                    }
                                    System.out.println("This is after second block.");
                          }
            }
    }
```

The output would be –

```
    Before the break
    This is after second block
```

As we can see in the above program, the usage of ***break*** with a label takes the control out of the second block directly.

Using continue

Sometimes, we may need to proceed towards next iteration in the loop by leaving some statements. In such situations, we can use ***continue*** statement within ***for, while*** and ***do-while***. For example –

```
for (int i=1; i<20;i++)
        if (i%2 == 0)
                continue;
        else
                System.out.println("i = " + i);
```

The above code snippet prints only the odd numbers in the range of 1 to 20.

Using return

The ***return*** statement is used to explicitely return the method. Based on some condition, we may need to go back to the calling method sometimes. So, we can use ***return*** in such situations.