3

Java Language Fundamentals

This chapter familiarizes readers with the fundamentals of Java programming language. The first part of the chapter deals with various data types in Java, literals, identifiers, operators and so on. The second part gives an overview of various control statements and arrays in Java.

3.1 The Building Blocks of Java

Java, as a programming language, follows a set of rules that define any programming language. There are predefined conventions that specify how *syntactically legal* constructs can be formed using the language elements. A *semantic definition* specifies the *meaning* of syntactically legal constructs.

3.1.1 Lexical tokens

Low-level language elements are called *lexical tokens*. These tokens are building blocks for more complex constructs. Identifiers, operators and special characters are all examples of tokens that can be used to build high-level constructs such as expressions, statements, methods and classes.

3.1.1.1 Identifiers

Identifiers are program elements (such as a variable, function/method, class and so on). Java has certain rules for naming identifiers. The naming rules are given below:

- Digits, letters, underscore or currency symbols ($, ¢, £ or ¥) are allowed
- The first letter of any identifier cannot be a digit.
- An identifier name must not be a reserved word
- Since Java programs are written in the Unicode character set, the definitions of letter and digit are interpreted according to this character set.
- Java is case-sensitive, label and Label are two different identifiers.

Legal Identifiers: number, Number, sum_$, average
Illegal Identifiers: 48sum, all/clear, end-to-end

3.1.1.2 Keywords

Programming languages have certain words, usually termed *reserved words* or *keywords*, that convey special meanings to the compiler. These keywords can be used only for their intended action and they cannot be used for any other purpose. Java has a richer set of keywords than C or C++.

The keywords in Java are given in Table 3.1.

Note that in addition to the keywords in the table, names of the built-in class, packages and interfaces are also words that cannot be used as identifiers.

3.1.2 Literals

In Java technology a literal denotes constant value. This implies that 0 is an integer literal and c is a character literal. Reserved literals true and false are used to represent boolean literals. 'A simple java program' is a string literal. In addition there is the null literal (null) which represents the null reference.

3.1.2.1 Integer Literals

Integer literals are the primary literals used in Java programming. Integer literals can be of decimal, hexadecimal and octal formats. Decimal (base 10) literals appear as ordinary numbers with no special notation. Hexadecimal (base 16) literals are prefixed with a leading 0x or 0X. Octal (base 8) literals appear with a leading 0 in front of the digits.

Table 3.1 Keywords in Java.

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

* It is important to note that const and goto are not currently in use.
* Also it should be remembered that Java language is case sensitive, so even though super is a keyword, Super is not. **All the Java technology keywords are in lower case.**

For example, the integer literal for 12 can be represented in the following ways:

- 12 *in decimal format*
- 0xC *in hexadecimal format*
- 014 *in octal format*

3.1.2.2 Floating-point literals

Floating-point literals represent decimal numbers with fractional parts, such as 3.145. They can be expressed in either standard or scientific notation, meaning that the number 563.84 can also be expressed as 5.6384e2. Unlike integer literals, floating-point literals default to the *double type*, which is a 64-bit value. Programmers can also use the smaller, 32-bit float type if they know that the full set of 64 bits is not required. This is achieved by appending an f or F to the end of the number, as in 5.6384e2f. The programmer can also explicitly state that he or she wants a *double type* as the storage unit for literal, as in 3.142d. However, this addition is not required as the default storage for floating-point numbers is already *double*.

3.1.2.3 Boolean Literals

Boolean literals are commonly used in Java programming. A reason for their frequent use is that they are present in almost every type of control structure. Boolean literals are needed to represent a condition or state that has only two possible values. Boolean values can be denoted using the reserved literals true or false.

3.1.2.4 Character Literals

Unicode is a standard universal character encoding representation. It uses two bytes (16-bit character set) and also represents all international languages as character sets in Java. Unicode is language-independent and is maintained by the Unicode Consortium. This encoding standard provides the basis for processing, storage and interchange of text data in any language in all modern software and information technology protocols.

Table 3.2 Special characters.

Description	Representation	Description	Representation
Backslash	\\	Newline	\n
Continuation	\	Single quote	\'
Backspace	\b	Double quote	\"
Carriage return	\r	Unicode character	\udddd
Form feed	\f	Octal character	\ddd
Horizontal tab	\t		

Character literals are represented by 16-bit Unicode character and appear within a pair of single quotation marks. An example of a Unicode character literal is \u0048, which is a hexadecimal representation of the character H. This same character is represented in octal as \110. Some special characters (control characters and characters that cannot be printed) are represented by a backslash (\) followed by the character code. A good example of a special character is \n, which forces the output to a new line when printed. Table 3.2. shows the special characters supported by Java.

3.1.2.5 String literals

A string literal is a sequence of characters, that are given within quotation marks and must occur on a single line. Escape sequences as well as Unicode values can appear in string literals:

Example 1

```
"This is a tab.\t This is a second one \u0009!"
```

The tab character is specified using the escape sequence and the Unicode value, respectively. The output is

```
This is a tab. This is a second one!
```

Example 2

```
"What's your option?"
```

The single apostrophe need not be specified using the escape sequence character in strings, but it would be if specified as a character literal (\').

The output is

```
What's your option?
```

Example 3

```
"\"String literals are double-quoted.\""
```

The double apostrophes in the string must be escaped using backslash (\).

The output is

```
"String literals are double-quoted."
```

Program 3.1 illustrates the use of string literals described above:

Program 3.1 Using string literals.

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

```
    {
        char ch = '\' ';
        String line1 = "This is a tab.\tThis is a second one \u0009!";
        String line2 = "What's your option?";
        String line3 = "\"String literals are double-quoted.\"";
        System.out.println(line1);

        System.out.println("A statement" +ch+ "s character literals are enclosed in
        single quotes");
        System.out.println (line2);
        System.out.println (line3);
    }
}
```

The output of Program 3.1 is the following:

```
This is a tab. This is a second one!
A statement's character literals are enclosed in single quotes
What's your option?
"String literals are double-quoted."
```

As character literals are enclosed in single quotes in the second println statement, an escape sequence (\) should be included in order to print a single quote as a character literal. In the third println statement, the string is delimited by double quotes, thus there is no need of escape sequence (\) for single quote.

3.1.3 White spaces

A white space is a sequence of spaces, tabs, form feeds and line-terminator characters. Line terminators can be newline, carriage return or carriage return–newline sequences in a Java source file. A Java program is a free-format sequence of characters, which is *tokenized* by the compiler, that is, broken into a stream of tokens for further analysis. Separators and operators help to distinguish tokens, but sometimes white spaces have to be inserted explicitly. Table 3.3 explains how different white spaces can be generated in a program.

For example, the identifier *classroom* will be interpreted as a single token unless a white space is inserted to distinguish the keyword class from the identifier *room*. White space aids not only in separating tokens, but also in formatting the program so that it is easy for programmers to read. The compiler ignores the white spaces once the tokens are identified.

Table 3.3 Types of white spaces.

White Space	Representation	No of Spaces / cursor position
Space	' '	single space
Tab	'\t'	4 spaces
New-Line	'\n'	Starting position in the newline
Carriage-return	'\r' or '\0'	Current position in the new line

3.1.4 Comments

A program can be documented by inserting comments at relevant places. The Java compiler ignores these comments. Java provides three types of comments:

- *Single-line comment* All characters that follow // until the end of the line constitute a single-line comment. For example,

```
// This comment ends at the end of this line.
```

- *Multiple-line comment* A multiple-line comment, as the name suggests, can span several lines. Such a comment starts with /* and ends with */. For example,

```
/* A comment on several
lines.
*/
```

- *Documentation comment* A documentation comment is a special purpose comment which when placed at appropriate places in the program can be extracted and used by the javadoc utility to generate HTML documentation for the program. Documentation comments are usually placed in front of class, interface, method and variable definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with /** and ends with */. For example,

```
/**
* This class implements open an account
* @author PRRao.
* @version 1.0
*/
```

Comments cannot be nested. The comment-start sequences (//, /*, /**) are not treated differently from other characters when occurring within comments.

3.2 Data Types

A data type defines a set of values and the operations that can be performed on them.

Unlike C++, Java is a strongly typed language.

There are seven primitive data types. Some details of these are given in Table 3.4. For example, short occupies two bytes (16-bits) and hence the minimum value is $-2^{15} = -32,768$ and the maximum value is $2^{15} - 1 = 32,767$. The generalized formula for minimum value is $-2^{(n-1)}$ and for maximum value is $2^{(n-1)} - 1$, where n is the number of bits.

Data types in Java can be divided into three main categories: integral, floating point and boolean. These types are described in the following sections.

3.2.1 Integral data types

Integral data types consist of integers and characters: of these, integers are described as follows.

3.2.1.1 Integers

Integer data types are byte, short, int and long. They represent *signed integers*.

Some examples of defining integer data types are given below.

- int i = – 215
- int max = 0x7fffffff (hexadecimal integer literal with a value of 2,147,483,647)
- int min = 0x80000000 (hexadecimal integer literal with a value of –2,147,483,648)
- long distance = 05402202647L (octal long literal)
- long zipcode = 55584152L (long literal)

3.2.1.2 Characters

In Java, the character data type is represented by char. Unicode character encoding is used by Java. The unicode standard was selected because it would help in internationalization of Java as

Table 3.4 Primitive data types, their widths and ranges.

Data Types	Width (in bytes)	Minimum Value	Maximum Value
byte	1	-2^7	$2^7 - 1$
short	2	-2^{15}	$2^{15} - 1$
int	4	-2^{31}	$2^{31} - 1$
long	8	-2^{63}	$2^{63} - 1$
char	2	0x0	0xffff
float	4	$1.401298e^{-45}$	$3.402823e^{+38}$
double	8	$3.940656e^{-324}$	$1.797693e^{+308}$

a programming language. The char data type represents symbols in the Unicode character set, such as letters, digits and special characters. It encompasses all the 65536 (2^{16}) characters in the Unicode character set which are written in the form of 16-bit values. The first 128 characters of the Unicode set (0–127) are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set (0–255) correspond to the 256 characters of the 8-bit ISO Latin-1 character set.

3.2.2 Floating-point numbers

This category includes the data types float and double. They represent fractional and signed numbers.

Examples of these are the following:

- float pi = 3.14159F
- double p = 313.159e–2
- double fraction = 1.0/3.0

3.2.3 Boolean data types

Boolean data type is used to represent logical values that can be either true or false. All relational, conditional and logical operators return boolean values. They are primarily used to keep track of the flow during program execution.

At this point it is essential to note that boolean values cannot be converted to other primitive data values, and vice versa. Primitive data values are *atomic* and are not *objects*. Each primitive data type defines the range of values in the data type and special operators in the language define operations on these values. Each primitive data type has a corresponding wrapper class that can be used to represent a primitive value as an object.

Program 3.2 demonstrates the use of various data types.

Program 3.2 Using different data types.

```
class Testdatatypes
{
    public static void main(String[ ] args)
    {                          // Range of various data types
      byte b = 10;             // –128 to 127
      short s = 45;            // –32,768 to 32,767
      int x = 50;              // –2,147,483,648 to 2,147,483,647
      long l1 = 2359878        // –9,223,372,036,854,775,808 to
                                                  9,223,372,036,854,775,807
      float f =3.142f;         // 3.4e–038 to 3.4e+038
      double d =1.98e8;        // 1.7e–308 to 1.7e+308
```

```
        char ch = 'a';          // 0 to 65,536
        boolean bo = false; // true or false
        System.out.println(b);
        System.out.println(s);
        System.out.println(x);
        System.out.println(l1);
        System.out.println(f);
        System.out.println(d);
        System.out.println(ch);
        System.out.println(bo);
    }
}
```

3.3 Variable Declarations

3.3.1 Declaring, initializing and using variables

Variables are defined by giving the name of the data types before the variable name. In Java, variables must be defined prior to their use. A variable stores values of data types. A variable has a particular size and a value associated with it. In Java, variables can only store values of primitive data types and references to objects. For example,

```
int i;
double d;
char c;
```

Several variables of the same type can also be declared in the same line; however, they need to be separated with a comma (,). For example,

```
int x, y, z;
double d, f;
```

Variables can also be initialized at the time of data type declaration, as in the following examples.

```
int i =10;
int x = 10, y =20;
char c, st1 = 'A';
long big = 2147483648L;
```

3.3.2 Variable types in Java

Java defines the three types of variable given below. Each variable can store values of primitive data types or references to objects only.

Instance variables These variables are members of a class and are instantiated for each object of the class. In other words, all instances, that is, objects, of the class will have their own instances of these variables, which are local to the object. The values of these variables at any given time constitute the state of the object. The data type should be specified explicitly for a variable at the time of declaration.

Static variables or class variables These variables are also members of a class, but these are not instantiated for each object of the class and therefore belong only to the class.

When a variable is declared static, it can be accessed before any objects of its class are created, without reference to any objects. Instance variables declared as static are, essentially, global variables when objects of its class are declared; no copy is made of a static variable. Instead, all instances of the class share the same static variable.

It is illegal to refer to any instance variable inside a static method.

Program 3.3 explains the use of instance and static variables.

Program 3.3 Using instance and static variables.

```
class TestStaticInst
{
    int age =12;
    String name= "Hyderabad";        //Instance variable
    static int a=34;                 //static variable
    static int b;

    static void smethod(int x)
    {
        System.out.println("x = "+x);
        System.out.println( "a= "+a);
        System.out.println("b= "+b);
    }

    public void mmethod()
    {
        System.out.println("Name and age are Instance variable"+ name +age);
    }
    static
    {
        System.out.println("static block initialized");
```

```
        b=a+2;
    }
    public static void main(String[ ] args)
    {
        TestStaticInst p1 = new TestStaticInst ();
                                    //p1 is a variable of type TestStaticInst class.
        p1.mmethod();
        smethod(32);
    }
}
```

The Output of the program is given below

```
static block initialized
Name and age are Instance variable Hyderabad12
x = 32
a = 34
b = 36
```

Local variables (method automatic variables) These variables are declared in methods and in blocks and are instantiated for each invocation of the method or block. In Java, local variables must be declared before they can be used.

A local variable is defined inside a method and is only visible from within the method. When execution of the method starts, the variable is available, and when the method ends the variable does not exist. For example, in the following code,

```
class T1
{
    static int x;
    public static void main(String a[])
    {
        int x=(x=2)*2;
        System.out.println(x);
    }
}
```

Here the local variable x is assigned before it is used and hence prints 4 as output.

3.3.3 Object reference variables

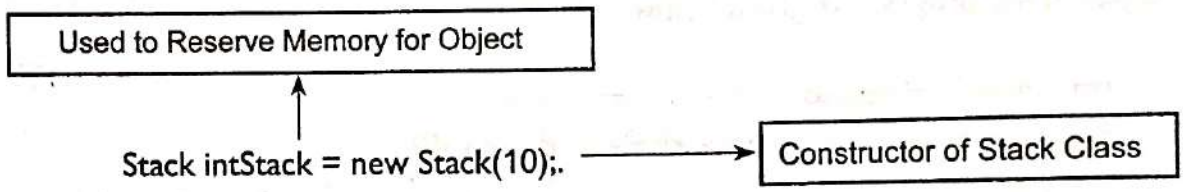
An object reference provides a *handle* for an object. These references can be stored in variables. In Java, reference variables must be declared and initialized before they can be used. A reference

variable has a name and a data type or class associated with it. A reference variable declaration, in its simplest form, can be used to specify the name and the data type. This determines the object a reference variable can denote. For example,

```
Stack intStack;
```

The above declaration enables variable intStack to reference objects belonging to the class Stack. It is important to note that the above declaration will not create *objects* of class Stack. They only create *variables*, which can store references to objects of these classes. A declaration can also include an initializer to create an object that can be assigned to the reference variable. (Constructors will be explained in chapter 4.)

Example



Now the reference variable intStack can reference objects of class Stack. The keyword new together with the constructor Stack (10) creates an object of class Stack. The reference to this object is assigned to the variable intStack. The newly created object of class Stack can now be manipulated through the reference stored in this variable. For example, refer to the following code:

```
class ml
{
    public static void main(String a[ ])
    {
        String name= new String("Hyderabad");
        int num=30;
        name= name+"city";
        num=num+ 30;
        System.out.println (name + num);
    }
}
```

Here, lines 5 and 6 indicate how an object and a reference variable are created. Lines 7 and 8 indicate how they are manipulated.

3.3.4 Default values for member variables

- If static variables in a class are not explicitly initialized, then when the class is loaded they are initialized to default values.
- Instance variables are also initialized to default values when the class is instantiated, unless they are explicitly initialized to some other value.

- A reference variable is initialized to the value *null*.
- Local variables are not initialized.

For example, see the following cases.

```
class Stack
{
    // Static variable
    static int counter; // Default value 0 when class is loaded.

    // Instance variables
    int size = 100; // Explicitly set to 100.

    Vector stack; // Implicitly set to default value null.
}
```

The default values of variables of different data types are given in Table 3.5.

3.3.5 Initializing local variables of different data types

Local variables are *not* initialized when they are instantiated at the time of method invocation. The javac compiler identifies local variables that are not initialized. Program 3.4 gives examples of uninitialized local variables.

Program 3.4 Example of uninitialized local variables.

```
public class SampleClass
{
    public static void main(String args[ ])
    {
        int weight = 10, price; // local variables
        if (weight < 10) price = 100;
        if (weight > 50) price = 5000;
        if (weight >= 10) price = weight*10; // Always executed.
        System.out.println("The price is:" + price);
    }
}
```

In Program 3.4, the compiler reports that the local variable price in the println statement is initialized; however, the last if-statement in the program assigns a value of 100 to the local variable price before it is used in the println statement. The compiler does not perform a rigorous analysis of the program in this regard. The program will compile correctly if the variable was initialized in the declaration, or if an unconditional assignment is made to the variable in the method.

Table 3.5 Default values of variables of different types.

Data Type	Default Value
Boolean	false
Char	'\u0000'
integer (byte, short, int, long)	0
floating point (float, double)	+0.0F or +0.0D
object reference	null

3.3.6 Initializing local reference variables

Initialization rules that apply to local variables also apply to local reference variables. Program 3.5A gives an example of initializing local reference variables.

Program 3.5A Example of uninitialized local reference variables.

```
public class SimpleClass
{
    public static void main(String args[ ])
    {
        String line; // local reference variable
        System.out.println("The string length is:" + line.length());
    }
}
```

In Program 3.5A, the compiler reports that the local variable line in the println statement may not be initialized, with a statement 'variable line might not have been initialized'. Objects should be created and their state initialized appropriately before use. If the variable line is set to the value null, the program will compile without any problems.

However, at run-time, an exception (NullPointerException) will be thrown, since the variable line will not reference any object. The golden rule is to ensure that a reference variable denotes an object before invoking methods via the reference. It must be ensured that the variable is not null. Program 3.5B is an example where local reference variables have been initialized.

Program 3.5B Example of initialized local reference variables.

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

```
    {
        String line= "reference variable";
        System.out.println("The String length is:"+ line.length());
    }
}
```

The output of Program 3.5B is

```
The String length is: 18
```

3.4 Wrapper Classes

Each primitive data type in Java has a corresponding wrapper class. Wrapper classes are used to represent a primitive data type as an object. In other words, a wrapper class is used to convert a primitive data type into its equivalent class type. Wrapper classes belong to the java.lang package. These wrapper classes also define useful methods for manipulating both primitive data values and objects. Table 3.6 gives the list of wrapper classes.

> Wrapper classes for integers and floating-point numbers are sub-classes of the Number class in the java.lang package.

Program 3.6 illustrates how to use wrapper classes (primitive double to wrapper Double and vice versa)

Program 3.6 Using wrapper classes.

```
import java.util.ArrayList;
public class TestWrapper
{
    public static void main(String[ ] args)
    {
        double dValue = 345.876;
        double primResult = objNegate(dValue);
        System.outprintln("Converting double into Double" + " " + prim.Result);
        primResult = primNegate(dValue);
        System.outprintln("Converting Double into double"+ " " + prim Result);
        Double objValue = –23.567;
        Double objResult = primNegate(objValue);
        System.outprintln("Converting Double into double"+ " " + objResult);
```

Table 3.6 Wrapper classes for the primitive data types.

Primitive data type	Wrapper classes	Primitive data type	Wrapper classes
int	Integer	long	Long
byte	Byte	float	Float
short	Short	double	Double

```
        objResult = objNegate(objValue);
        System.outprintln("Converting double into Double" + " " + objResult);
    }
    public static Double objNegate(Double n)
    {
        return n;
    }
    public static double primNegate(double n)
    {
        return –n;
    }
}
```

The output of Program 3.6 is given below:

```
Converting double into Double  345.876
Converting Double into double  –345.876
Converting Double into double  23.567
Converting double into Double  –23.567
```

3.5 Operators and Assignment

3.5.1 Operators

An operator takes one or more arguments and produces a new value. Arguments are distinct from ordinary method calls, but the effect is one and the same. Some of the commonly used Java technology operators are given in Table 3.7.

In Java, most of the time, manipulations are done only on primitive data types; however, there are some exceptions, such as =, == and !=. These operators work with all objects. In addition, the String class supports + and += operators.

Table 3.7 Commonly used operators in Java.

Functions	Operator	Functions	Operator
Multiplication	*	Assignment	=
Addition	+	Left shift	<<
Subtraction	–	Right shift	>> and >>>
Logical and	&&	Equality comparison	= =
Conditional Operator	? :	Non-equality comparison	!=

3.5.2 Assignment

In Java, = represents the assignment operator. This operator is used to assign values of entities on the right-hand side to the variable on the left-hand side (example $a = b$). In Java, the attribute to the left of the assignment operator (in this case a) is a distinct variable. The attribute on the right of the assignment operator (in this case b) can be any constant, variable or expression that can produce a value. For instance, you can assign a constant value to a variable with the statement $a = 3$.

Assignment of primitive data types is quite easy, because primitive types hold an actual values (and not handles to some object). When primitives are assigned, contents of one variable are copied to another. For example, if $x = y$ for primitives, then the contents of y are copied into x. If x is modified, y remains unaffected by this modification.

Program 3.7 illustrates the use of the assignment operator:

Program 3.7 The assignment operator.

```
class Value
{
    public int i =10;
}
class Assignment
{
    public static void main(String[ ] args)
    {
        Value v1 = new Value();
        Value v2 = new Value();
        Value v3 = new Value();
```

```
        v1.i = 22;
        v2.i = 30;
        System.out.println("v1.i:" + v1.i + ", v2.i:" + v2.i+ ", v3.i:" + v3.i);
        v2 = v1;
        v3.i = 37;
        System.out.println("v1.i:" + v1.i + ", v2.i:" + v2.i + ", v3.i:" + v3.i);
        v2 = v3;
        System.out.println("v1.i:" + v1.i + ", v2.i:" + v2.i+ ", v3.i:" + v3.i);
    }
}
```

In the above program, two classes are declared (Value and Assignment). The Value class declares an integer variable *i* which is initialized to 10. Three instances of the Number class (v1, v2 and v3) are created within main() method of the Assignment class.

The *i* value within each Value is given a different value, and then v1 is assigned to v2, v3 is assigned to v2 now v3, v2 will change. Changing the v1 object appears to change the v2 object and also changing v3 object appears to change the v2 object. The output of the program is the following:

```
v1.i: 22, v2.i: 30, v3.i: 10
v1.i: 22, v2.i: 22, v3.i: 37
v1.i: 22, v2.i: 37, v3.i: 37
```

Assignment operators actually work with all the fundamental data types. Table 3.8 lists the assignment operators.

3.5.2.1 Conversion rules in assignments

There are some basic conversion rules for assignment when the source and destination are of different types.

Table 3.8 Assignment operators.

Description	Operator	Description	Operator
Simple	=	Modulus	%=
Addition	+=	AND	&=
Subtraction	-=	OR	|=
Multiplication	*=	XOR	^=
Division	/=		

If the source and destination are of the same type they can be assigned without any issues. As for example, in the following cases.

```
int i = 5;
int j = 12;
i = j; // Valid, since both are of same type
```

Widening If the source is of smaller size than the destination but the source and destination are of compatible types, then no casting is required. An *implicit widening* takes place in this case. An example is assigning an int to a long.

```
int i=5;
long l=12;
l = i;          // Valid, both are of compatible types and source is of smaller size
```

Casting If the source and destination are of compatible types, but the source is of larger size than the destination, an *explicit casting* is required. In this case, if no casting is provided then the program does not compile.

```
int i=5;
long l=12;
i = (int) l;    // Valid, both are of compatible types and source is of larger size
                // and hence explicitly cast.
```

For more details on casting see section 3.5.12.

3.5.3 Mathematical operators

The basic mathematical operators used in Java are the same as the ones available in most programming languages: addition (+), subtraction (–), division (/), multiplication (*) and the modulus operator (%), which *produces the remainder* from integer division. Integer division truncates, rather than rounding off the result.

Java also uses a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language. For example, to add 4 to the variable *x* and assign the result to x, use: x += 4. Table 3.9 gives a description of the different types of arithmetic operators.

Table 3.9 Arithmetic operator.

Description	Operator	Description	Operator
Addition	+	Division	/
Subtraction	–	Modulus	%
Multiplication	*		

3.5.3.1 Unary minus and plus operators

The unary minus (–) and unary plus (+) are the same operators as binary minus and plus. The compiler understands which use is intended by the way you write the expression.

For instance, the statement

```
Average = –5;
```

assigns the value – 5 to the variable *Average*.

```
x = a * –b;
```

This may confuse the reader. A clearer way to write this statement is

```
x = a * (–b);
```

The unary minus produces the negative of the variable it acts on. The unary plus provides symmetry with the unary minus, although it does not do anything significant. Table 3.10 gives a description of the unary operators.

3.5.3.2 Auto increment and decrement

Java provides two operators to increment and decrement a unit. The decrement operator is `--` and means, 'decrease by one unit'. The increment operator is `++` and means 'increase by one unit'. For example, the expression `++a` is equivalent to saying `a = a + 1`. There are two versions of each type of operator, often called the prefix and postfix versions. For preincrement and predecrement, (`++A` or `--A`), the operation is performed and the value is produced. For postincrement and postdecrement (`A++` or `A--`), the value is produced, then the operation is performed.

Table 3.10 Unary operators.

Description	Operator	Description	Operator
Increment	++	Negation	-
Decrement	--	Bitwise complement	~

Program 3.8 demonstrates Unary Operators:

Program 3.8 Action of unary operators.

```
class IncDec
{
    public static void main (String args[])
    {
        int x = 8, y = 13;
```

```
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("++x = " + ++x);
        System.out.println("y++ = " + y++);
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

Program 3.8 produces the following results:

```
x = 8
y = 13
++x = 9
y++ = 13
x = 9
y = 14
```

3.5.4 Relational operators

Relational operators generate a boolean result. These operators evaluate the relationship between the values of the operands. The value true is given if the relationship is true, and the value false if the relationship is untrue. The relational operators are *less than* <, *greater than* >, *less than or equal to* <=, *greater than or equal to* >=, *equivalent* == and *not equivalent* !=. Table 3.11 gives a description of relational integer operators.

The equivalence and non-equivalence operators work with all built-in data types. Other comparison operators will not work with boolean types.

3.5.4.1 Relational operators and objects

The relational operators == and != also work with all objects. Program 3.9 illustrates how they may be used to test the equivalence of two objects.

Table 3.11 Relational integer operators.

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

Program 3.9 Testing object equivalence.

```
public class Equivalence
{
    public static void main(String[ ] args)
    {
        Integer n1 = new Integer(22);
        Integer n2 = new Integer(22);
        System.out.println(n1 = = n2);
        System.out.println(n1 != n2);
    }
}
```

In the above code, the result of the boolean comparison in the expression System.out.println (n1 = = n2) will be given as output. In this case, naively, we may expect the outputs to be true and then false, since both Integer objects, n1 and n2, appear the same; however, despite the fact that the contents of the objects are the same, their handles are not the same. The operators == and != compare object handles. Hence, the output is actually false and then true.

To compare the contents of an object for equivalence, Java provides a special method called equals(). This method can be used in all objects excluding primitives types, which work fine with == and !=. This method is illustrated in Program 3.10.

Program 3.10 Use of the equals() method.

```
public class EqualsMethod
{
    public static void main(String[ ] args)
    {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
    }
}
```

The output of the above program will be true, as is expected; however using the equals() method again has certain bottlenecks. For example, consider Program 3.11.

Program 3.11 Difficulties in using the equals() method.

```
class Value
{
    int i;
}
public class EqualsMethod2
{
    public static void main(String[ ] args)
    {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = 47;
        v2.i = 47;
        System.out.println(v1.equals(v2));
    }
}
```

Here, v1 and v2 are handles for the instance variable *i*.

The output generated by Program 3.11 would be false. The reason for such an output is that the default behaviour of equals() is to compare handles. Therefore, the programmer needs to override (give a different functionality to) the equals() method in a new class to obtain the desired behaviour. Most of the Java library classes implement the equals() method so that it can compare the contents of objects instead of their handles. The concept of overriding is discussed in chapter 4.

3.5.5 Logical operators

Logical operators AND (&&), OR (||) and NOT (!) produce a boolean value of true or false based on the logical relationship of its arguments. These operators are applied to boolean values only. Java prevents the use of non-boolean values in a logical expression, contrary to the fact that this is allowed in C and C++. Subsequent expressions, however, produce boolean values using relational comparisons, instead of using logical operations on results.

The following illustration helps to understand the use of logical operators: let $i = 85$ and $j = 4$; then the various logical operators, acting on i and j give the following results.

$i > j$ is true	$i < j$ is false
$i >= j$ is true	$i <= j$ is false
$i == j$ is false	$i != j$ is true
$(i < 10)$ && $(j < 10)$ is false	$(i < 10)$ || $(j > 10)$ is true

In Java, a boolean value is automatically converted to an appropriate text form if it is used where a String is expected. In the above illustration, the primitive data type that is used can be

replaced with any other primitive data type except boolean type; however, caution should be taken during the comparison of floating-point numbers.

3.5.6 Bitwise operators

Java declares a variety of bitwise operators. Programmers use these operators to manipulate integer data types (int, long, short and byte) and the char type. These operators perform boolean algebra on the corresponding bits in their arguments to generate the result. Table 3.12 gives a description of bitwise and shift operators.

In order to unite/combine assignment and operation, bitwise operators are joined with the = sign. Legal operators include &=, |= and ^=. (It should be noted that as ~ is a unary operator it cannot be combined with the = sign.)

In case of boolean data type, which is treated as a 1-bit value and so is somewhat different, programmers can perform bitwise AND, OR and XOR operations. This is because a boolean data type is a 1-bit value. The bitwise NOT operation cannot be performed on boolean types.

3.5.7 Shift operators

Java also provides shift operators to manipulate bits. These operators can be used only with primitive integral types. The left-shift operator (<<) acts on the operand to the left of the operator and produces a result shifted to the left by the number of bits specified after the operator (inserting zeros at the lower-order bits). The signed right-shift operator (>>) acts on the operand to the left of the operator and produces a result shifted to the right by the number of bits specified after the operator. The signed right shift >> uses sign extension: if the value is positive, zeros are inserted

Table 3.12 Bitwise and shift operators.

Description	Operator	Results
Bitwise AND	&	Gives output 1 if both input bits are 1, else 0
Bitwise OR	|	Gives output 1 if at-least one of the input bit is 1, else 0
Bitwise XOR	^	Gives output 1 if anyone input bit is 1 but not both, else 0
Bitwise NOT	!	Gives output 1 if input bit is 1 and output 1 if input bit is 0
Left-shift	<<	Shifts the bits of operand to left by number of bits specified and insert zeros at lower order bits.
Right-shift	>>	Shifts the bits of operand to right by number of bits specified. If operand is positive, zeros are inserted at the higher order bits, else ones are inserted at the higher order bits.
Zero-fill-right-shift	>>>	It is unsigned right shift. Shifts the bits of operand to right by number of bits specified and insert zeros at lower order bit irrespective of the sign of the operand.

at the higher-order bits; if the value is negative, ones are inserted at the higher-order bits. Java has also added the unsigned right shift >>>, which uses zero extension: regardless of the sign of the operand, zeros are inserted at the higher-order bits.

Suppose x = 3, the 8-bit representation of 3 is 00000011

For x<<2, the result is 12, since it shifts 1 bits 2 positions to right as shown below.

Shifted 0 0|0 0 0 0 1 1 → 0 0 0 0 1 1 0 0 → 8+4 → 12
$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

For x>>2, the result is 0, since it shifts 1 bits 2 positions to left as shown below.

x>>2
Shifted 0 0 0 0 0 0|1 1 → 0 0 0 0 0 0 0 0 → 0
$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

Similarly, the results for x<<3 and x>>3 are given below:

x<<3
Shifted 0 0 0|0 0 0 1 1 → 0 0 0 1 1 0 0 0 → 16+8 → 24
$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

x>>3
Shifted 0 0 0 0 0|0 1 1 → 0 0 0 0 0 0 0 0 → 0
$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

A single left-shift for an unsigned number is equivalent to multiplication by 2. Similarly, a single right-shift for an unsigned number is equivalent to division by 2.

3.5.8 Ternary If–Else operator

This operator is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary if–else statement. The expression is of the form

```
boolean-exp ? value0 : value1
```

For example, when calculating the maximum of three numbers:

d = (a>b)? (a>c? a:c): (b>c? b: c); where a, b, c, d are numerical values.

If the values of a, b, c are 4, 5, 6, respectively, then, first a, b are compared; that is (4 > 5). This result is false, so the control jumps to the condition b > c, that is (5 > 6), whose result is again false. Hence the final result is c, that is 6.

If boolean-exp evaluates to true, value0 is evaluated and its result becomes the value produced by the operator. If boolean-exp is false, value1 is evaluated and its result becomes the value produced by the operator.

3.5.9 Comma operator

The comma is used in C and C++ not only as a separator in function argument lists but also as an operator for sequential evaluation. The sole usage of the comma operator in Java is in for loops.

3.5.10 String operator +

The + operator in Java can be used to concatenate strings. The use of String + is interesting. If an expression begins with a String, then all operands that follow must be strings:

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

The execution of the above statements prints the result x, y, z012.

Here, the Java compiler will convert x, y and z into their String representations instead of adding them together first; however, the code can be written as shown below:

```
System.out.println(x + sString);
```

Then earlier versions of Java will signal an error, later versions, however, will turn x into a String. In case a String (using an earlier version of Java) is being concatenated, it should be ensured that the first element is a String (or a quoted sequence of characters which the compiler recognizes as a String).

3.5.11 Operator precedence

Operator precedence defines how an expression evaluates when several operators are present. Java has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before addition and subtraction. Programmers often forget the other precedence rules, so the use of parentheses is recommended to make the order of evaluation explicit.

For example:

```
a = x + y – 2/2 + z;
```

has a very different meaning from the same statement with a particular grouping of parentheses as given below:

```
a = x + (y – 2)/(2 + z);
```

In the first case, a = x+y – 2/2 + z, the equation must be solved according to the precedence of the operators. The higher priority operator is executed first, followed by the lower priority operators. Now, for example, let us consider that the values of x, y and z are 1, 2 and 3, respectively. As the division operator has the highest priority it will be executed first. So now the equation is 1 + 2 – 1 + 3. The priority of addition and subtraction is the same, so the left associativity

principle applies here, and hence, first addition then subtraction and then addition operations will be executed. Hence the final result after addition is 5. In the second case, a = x + (y – 2) /(2 + z), let us consider that the values of x, y and z are 1, 4 and 1, respectively. As the parenthesis has higher precedence than division, this time parenthesis will be executed first. So, the equation is 1 + (4 – 2)/(2 – 1) whose result after execution of brackets is 1 + 2/1. Next we execute the division operator. The result is 1 + 2. Hence, the final result after execution of addition operator is 3.

The following is a list of all Java operators from highest to lowest precedence. In this list of operators, all the operators in a particular row have equal precedence. The precedence level of each row decreases from top to bottom. This means that the [ ] operator has a higher precedence than the * operator, but the same precedence as the ( ) operator. Table 3.13 gives a list of operators and the manner in which each takes precedence over others.

Evaluation of expressions still moves from left to right, but only when dealing with operators that have the same precedence. Otherwise, operators with a higher precedence are evaluated before operators with a lower precedence.

3.5.12 Casting operators

In Java one type of data is changed automatically into another type when appropriate. For instance, if an integral value is assigned to a floating point variable, the compiler will automatically convert the integer value into a float value. *Casting* allows programmers to make this type conversion explicit, or to force it when it would not normally happen.

Table 3.13 Precedence rules of operators in Java.

Level	Operators on each level			
1	.	[]	()	
2	++	--	!	~
3	*	/	%	
4	+	-		
5	<<	>>	>>>	
6	<	>	<=	>=
7	==	!=		
8	&			
9	^			
10	&&			
11	||			
12	?:			
13	=			

To perform a cast, put the desired data type (including all modifiers) inside parentheses, to the left of any value. Here is an example of how easting can be used in a program:

```
void casts()
{
  int a = 100;
  long x = (long)a;
  long y = (long)100;
}
```

From the above example, it is clear that a cast can be performed on a numeric value as well as on a variable. The casting illustrated here, however, is superfluous, since the compiler will automatically promote an int value to a long whenever necessary. The programmer can still use a cast in order to state the point explicitly or to make the code clearer. In other situations, a cast is essential just to get the code to compile.

In Java, casting is safe, with the exception that when a narrowing conversion (casting a data type that can hold more information into one that cannot) is performed there is a risk of losing information. In the case of widening conversion, an explicit cast is not needed because the new type will do more than hold the information from the old type so that no information is ever lost.

Java allows casting of any primitive type to any other primitive type (except for boolean). Class types do not allow casting. To convert one to the other there must be special methods.

3.6 Control Structures

The ability of a computer program to perform complex tasks is built on just a few ways of combining simple commands into control structures. The six control structures are block, the while loop, the do...while loop, the for loop, the if statement, and the switch statement. Each of these structures is considered to be a single 'statement', but each is in fact a *structured* statement that can contain one or more other statements inside it.

3.6.1 Block

Block is the simplest type of structured statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is as follows:

```
{
    statements
}
```

A block consists of a sequence of statements enclosed between the pair of flower brackets, '{' and '}'. Block statements usually occur inside other statements, where their purpose is to group together several statements into a unit; however, a block can be legally used wherever a statement can occur.

Two examples of the use of blocks are given in Programs 3.12(a) and (b):

Program 3.12 (a) simple block; (b) a block with a variable defined within it.

(a) Simple block

```
{
    System.out.print("The answer is");
    System.out.println(ans);
}
```

(b) A block containing a newly defined variable

```
{
    int temp;     // A temporary variable for use in this block.
    temp = x;     // Save a copy of the value of x in temp.
    x = y;        // Copy the value of y into x.
    y = temp;     // Copy the value of temp into y.
}
```

In the second example, a variable temp is declared inside the block. This is perfectly legal, and it is good style to declare a variable inside a block if that variable is used nowhere else but inside the block. A variable declared inside a block is completely inaccessible and invisible from outside that block.

Java defines five other control structures which can be divided into two categories: *loop statements* and *branching statements*.

3.6.2 Loop

Loop statements are those that refer to the while and do...while statements.

3.6.2.1 The while loop

A while loop is used to repeat a given statement over and over. Usually, a while loop will repeat a statement over and over, but only so long as a specified condition remains true.

```
while (boolean-expression)
        statement
```

Since the statement can be, and usually is, a block, many while loops have the form:

```
while (boolean-expression)
{
    statements
}
```

The above statements are explained in the example given below.

Example

```
int x= 1;
while (x<=5)// where x<=5 is a boolean expression
{
    x++;
    System.out.println (x);
}
```

Statement

The semantics of this statement go like this, When the execution comes to a while statement, it evaluates the boolean-expression, which yields either true or false as the value. If the value is false, the computer skips over the rest of the while loop and proceeds to the next command in the program. If the value of the expression is true, the computer executes the statement or block of statements inside the loop. Then it returns to the beginning of the while loop and repeats the process. This process will continue until the value of the expression is false, if that never happens, then there will be an infinite loop.

The compiler will not detect the infinite loop and hence the process will continue forever; to stop this process, the user has to perform abnormal termination using *Ctrl+Break* buttons.

Program 3.13 is an illustration of how to print five numbers:

Program 3.13 Printing numbers from 1 to 5.

```
class WhileExample
{
    public static void main(String args[ ])
    {
        int number;.
        number = 1;.
        while ( number < 6 ) {
        System.out.println(number);
        number = number + 1;
    }
}
```

In the above program, the variable number is initialized with the value 1. So the first time through the while loop, when the computer evaluates the expression number < 6, it is asking whether 1 is less than 6, which is true. The computer therefore proceeds to execute the two statements inside the loop. The first statement prints out '1'. The second statement adds 1 to number and stores the result back into the variable number, the value of number has been changed

to 2. The computer has reached the end of the loop, so it returns to the beginning and asks again whether number is less than 6. Once again this is true, so the computer executes the loop again, this time printing out 2 as the value of number and then changing the value of number to 3. It continues in this way until eventually number becomes equal to 6. At that point, the expression number < 6 evaluates to false. It should be noted that when the loop ends, the value of number is 6, but the last value that was printed was 5.

3.6.2.2 do...while statement

Sometimes it is more convenient to test the continuation condition at the end of a loop instead of at the beginning as is done in the while loop.

The do...while statement is very similar to the while statement, except that the word 'while,' along with the condition that it tests, has been moved to the end. The word 'do' is added to mark the beginning of the loop. A do...while statement has the form

```
do
    statement
while ( boolean-expression );
```

or, since, as usual, the statement can be a block,

```
do
{
    statements
} while ( boolean-expression );
```

Note the semicolon ';' at the end. This semicolon is part of the statement, just as the semicolon at the end of an assignment statement or declaration is part of the statement. Omitting it results in a syntax error.

To execute a do loop, the computer first executes the body of the loop, that is, the statement or statements inside the loop and then it evaluates the boolean expression. If the value of the expression is true, the execution returns to the beginning of the do loop and repeats the process, if the value is false, it ends the loop and continues with the next part of the program. Since the condition is not tested until the end of the loop, the body of a do loop is executed at least once.

3.6.2.3 The For statement

The for loop statement is almost similar to the while loop statement; however, for statements offer certain advantages over the while loop statements.

A for loop can be easier to construct and easier to read than the corresponding while loop.

It is quite possible that in real programs for loops actually outnumber while loops. A for statement makes a common type of while loop easier to write. while loops have the following general form:

```
initialization
while ( continuation-condition )
{
    statements
    update
}
```

Consider the example illustrated by Program 3.14 which calculates the total amount (principal + interest) repayable for a given rate of interest after 5 years.

Program 3.14 Calculation of interest using a while loop.

```
import java.io. *;
public class S1
{
    public static void main(String[ ] args) throws IOException
    {
        int interest,principal,years = 0,rate; // declaration and initializing variables
        System.out.println("Enter the principal amount and Rate");
        BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
        principal= Integer.parseInt(br.readLine());
        rate= Integer. parseInt(br.readLine());
        while ( years < 5 )
        {
            interest = principal * rate;        //
            principal += interest;              // do three statements
            System.out.println(principal);      //
            years++;                    // updates the value of the variable, years
        }
    }
}
```

The above program can be written more efficiently using the for statement as in Program 3.15:

Program 3.15 Calculation of interest using a for loop.

```
import java.io.*;
public class S1
```

```
{
    public static void main(String[ ] args) throws IOException
    {
        int interest,principal,years,rate;      // declaration and initializing variables
        System.out.println("Enter the principal amount and Rate");
        BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
        principal= Integer.parseInt(br.readLine());
        rate = Integer.parseInt(br.readLine());

        for( years=0;years< 5; years++ )
        {                                       // condition for continuing loop
            interest = principal * rate;        //
            principal += interest;              // do three statements
            System.out.println(principal);      //
        }
    }
}
```

The initialization, continuation condition and updating have all been combined in the first line of the for loop. This keeps everything involved in the 'control' of the loop in one place, which makes the loop easier to read and understand. A for loop statement is executed in exactly the same way as the original code. The initialization part is executed once before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition evaluates to false. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition.

The formal syntax of the for statement is as follows:

```
for ( initialization; continuation-condition; update )
    statement
```

If a block statement is used, it takes the following format:

```
for ( initialization; continuation-condition; update )
{
    statements
}
```

Here, **continuation-condition** must be a boolean-valued expression, whereas **initialization** can be any expression, as can **update**. Any of the three can be empty. If the continuation condition is empty, it is treated as if it were true, so the loop will be repeated forever or until it ends for some other reason, such as a break statement.

Some people like to begin an infinite loop with for (;;) instead of while (true).

Usually, the initialization part of a for statement assigns a value to some variable, and the update changes the value of that variable with an assignment statement or with an increment or decrement operation. The value of the variable is tested in the continuation condition, and the loop ends when this condition evaluates to false. A variable used in this way is called a loop control variable. In the for statement given above, the loop control variable is years.

Certainly, the most common type of for loop is the counting loop, where a loop control variable takes on all integer values between some minimum and some maximum value. A counting loop has the following form:

```
for ( variable = min;  variable <= max; variable++ )
{
    statements
}
```

where min and max are integer-valued expressions (usually constants); variable takes on the values min, min+1, min+2, ..., max. The value of the loop control variable is often used in the body of the loop. The for loop at the beginning of this section is a counting loop in which the loop control variable years takes on the values 1, 2, 3, 4, 5.

Here is an even simpler example in which the numbers 1, 2, ... 10 are displayed as a standard output:

```
for ( N = 1;  N <= 10;  N++ )
    System.out.println( N );
```

For various reasons, Java programmers like to start counting at 0 instead of 1 and they tend to use a '<' in the condition, rather than a '<='. The following variation of the above loop prints out the ten numbers 0, 1, 2, ... 9:

```
for ( N = 0 ;  N < 10 ;  N++ )
    System.out.println( N );
```

It is easy to count down from 10 to 1 instead of counting up. Just start with 10, decrement the loop control variable instead of incrementing it and continue as long as the variable is greater than or equal to one.

```
for ( N = 10 ;  N >= 1 ;  N-- )
    System.out.println( N );
```

The for statement also allows both the initialization part and the update part to consist of several expressions, separated by commas. For example, to count up from 1 to 10 and count down from 10 to 1 at the same time, we can write the for loop as given below.

```
for ( i=1, j=10;  i <= 10;  i++, j-- )
{
    TextIO.put(i,5);   // Output i in a 5-character wide column.
    TextIO.putln(j,5); // Output j in a 5-character column
}
```

The output of the program is:

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

Note that certain versions of Java do not support TextIO class. In such cases, the System.out.println command can be used.

The next step is to write a program losing a for loop to print even numbers between 2 and 20.

There are several ways to write this program. This is done in three different ways in Programs 3.16, 3.17 and 3.18.

Program 3.16 Printing even numbers from 2 to 20, method I.

```
// There are 10 numbers to print.
// Use a for loop to count 1, 2, ..., 10.  The numbers we want
// to print are 2*1, 2*2, ..., 2*10.

for (N = 1; N <= 10; N++)
{
    System.out.println( 2*N );
}
```

Program 3.17 Printing even numbers from 2 to 20, method II.

```
// Use a for loop that counts
// 2, 4, ..., 20 directly by
// adding 2 to N each time through the loop.

for (N = 2; N <= 20; N = N + 2)
{
    System.out.println( N );
}
```

Program 3.18 Printing even numbers from 2 to 20, method III.

```
// Count off all the numbers
// 2, 3, 4, ..., 19, 20, but only print out numbers that are even.
```

```
for (N = 2; N <= 20; N++)
{
    if ( N % 2 == 0 ) // is N even?
    System.out.println( N );
}
```

Perhaps it is worth stressing once more that a for statement, like any statement, never occurs on its own in a real program. The statement must be inside the main routine of a program or inside some other subroutine. And that subroutine must be defined inside a class.

Java also allows the use of char data types in a for statement.

```
for ( char ch = 'A';  ch <= 'Z';  ch++ )
{
    System.out.print(ch);
    System.out.println();
}
```

3.6.3 Branching statements

3.6.3.1 The if statement

The if statement tells the computer to take one of two alternative courses of action, depending on whether a given boolean-valued expression evaluates **true** or **false**. It is an example of a *branching* or *decision* statement. An if statement has the form given below:

```
if ( boolean-expression )
    statement1
else
    statement2
```

or with a block, as in the following structure:

```
if ( boolean-expression )
{
    statement1
}
else
{
    statement2
}
```

For example, consider the following statement:

```
if (number1 > number2)
    System.out.println (number1+"is greater than"+ number2);
```

If more than one statement should be executed when the condition is true, then these may be grouped together inside braces. For example,

```
if (number1 > number2)
{
    System.out.println (number1 + "is greater than"+ number2);
    System.out.println (number2 + "is less than or equal to"+ number1);
}
```

The if/else statement extends the idea of the if statement by specifying another section of code that should be executed only if the condition is false.

For example, in the following case,

```
if (number1 > number2)
    System.out.println (number1 + "is greater than" + number2);
else
    System.out.println (number1 + "is not greater than" + number2);
```

As in if selection structure, each of the statements may be compound statements consisting of more than one simple statement enclosed within braces.

It is also possible to make longer sequences as follows:

```
if (number1 > number2)
    System.out.println (number1 + "is greater than" + number2);
else
    if (number1 < number2)
        System.out.println (number1 + "is less than" + number2);
    else
        System.out.println (number1 + "is equal to" + number2);
```

When the if statement is executed, it evaluates the boolean expression. If the value is true, the computer executes statement1 and skips statement2 and vice versa. Note that at any point of time one and only one of the two statements inside the if statement is executed. The two statements represent alternative courses of action. The computer decides between these courses of action based on the value of the boolean expression.

Using only an if statement executes statements in its block if the boolean expression is true, otherwise the block is skipped and the next line is executed.

```
if ( boolean-expression )
    statement
```

or

```
if ( boolean-expression )
{
    statement
}
```

To execute these statements, the computer evaluates boolean-expression the expression. If the value is true, the computer executes **statement**, which is contained inside the if statement; if the value is false, the computer skips **statement** and moves to the next line of the program. Programs 3.19 and 3.20 illustrate the use of if and if-else statements.

Program 3.19 Using If statements to execute decisions in a program.

```
if ( x > y )
{
    int temp;     // A temporary variable for use in this block.
    temp = x;     // Save a copy of the value of x in temp.
    x = y;        // Copy the value of y into x.
    y = temp;     // Copy the value of temp into y.
}
```

Program 3.20 Using if–else statements.

```
if ( years > 1 ) {  // handle case for 2 or more years
    System.out.print("The value of the investment after");
    System.out.print(years);
    System.out.print("years is $");
}
else {  // handle case for 1 year
    System.out.print("The value of the investment after 1 year is $");
} // end of if statement
System.out.println(principal);  // this is done in any case
```

3.6.3.2 break and continue statements

The syntax of the while and do...while loops allows programmers to test the continuation condition at either the beginning of a loop or the end.

Sometimes, it is more natural to have the test in the middle of the loop, or to have several tests at different places in the same loop.

Java provides a general method for breaking out of the middle of any loop. It is called the break statement. This statement consists of just the following line:

```
break;
```

When the computer executes a break statement in a loop, it will immediately jump out of the loop. It then continues on to whatever follows the loop in the program. Consider, for example, the following fragment, Program 3.21:

Program 3.21 Using break to terminate a loop.

```
while (true) {  // looks like it will run forever!
    TextIO.put("Enter a positive number:");
    N = TextIO.getInnt();
    if (N > 0)   // input is OK; jump out of loop
       break;
    TextIO.putln("Your answer must be > 0.");
}
// continue here after break
```

If the number entered by the user is greater than zero, the break statement will be executed and the computer will jump out of the loop. Otherwise, the computer will print out Your answer must be > 0. and will jump back to the start of the loop to read another input value.

A break statement terminates the loop that immediately encloses the break statement. It is possible to have nested loops, where one loop statement is contained inside another.

> If you use a break statement inside a nested loop, it will only break out of that loop, not out of the loop that contains the nested loop.

The continue statement keeps executing from the beginning of the loop depending on the loop condition criteria. The following code illustrates the use of the continue statement

```
public class Testcontinue
{
    public static void main(String argv[])
    {
        for(int i=1;i<10;i++)
        {
            if (i>5) continue;
            System.outprint (i);
        }
    }
}
```

The output is 1 2 3 4 5.

3.6.3.3 switch statement

The switch statement provides a way to execute one of a number of different sections of code, according to the value of an integer or character variable. Each section of code is labelled with a case, which specifies the value(s) of the variable for which that section of code should be executed. The variable whose value to use is given at the top of the switch statement. For example, the following section of code prints out the roman numeral corresponding to any of the numbers from 0 to 5:

```
switch (number)
{
    case 1:
            System.out.println("I");
            break;
    case 2:
            System. out.println("II"); break;
    case 3:
            System.out.println("III");
            break;
    case 4:
            System. out. println("IV"); break;
    case 5:
            System.out.println("V");
            break;
    default:
            System.out.println ("?");
}
```

The break statement, which is used after each case, causes the program to skip the rest of the cases. Without the break statement, execution would continue through all the sections of code after the matching case. The last case is a default one. This will be executed if none of the other cases match.

Programmers can leave out one of the groups of statements entirely (including the break). In such cases they have two case labels in a row, containing two different constants. This just means that the execution will jump to the same place and perform the same action for each of the two constants. Program 3.22 illustrates the use of the switch statement.

Program 3.22 Using switch to construct multi-way branches.

```
switch (N)
{   // assume N is an integer variable
```

```
        case 1:
                System.out.println("The number is 1.");
                break;
        case 2:
        case 4:
        case 8:
                System.out.println("The number is 2, 4, or 8.");
                System.out.println("(That's a power of 2!)");
                break;
        case 3:
        case 6:
        case 9:
                System.out.println("The number is 3, 6, or 9.");
                System.out.println("(That's a multiple of 3!)");
                break;
        case 5:
                System.out.println("The number is 5.");
                break;
        default:
                System.out.println("The number is 7,");
                System.out.println("   or is outside the range 1 to 9.");
}
```

3.6.3.4 Empty statement

This is a statement that consists simply of a semicolon. Look at the following code:

```
if (x < 0)
{
    x = – x;
};
```

The semicolon after the } is legal; however, due to the presence of the semicolon the compiler considers it to be an empty statement, not part of the if statement.

3.6.3.5 Nested loops

Nested loops refer to loops within a loop. For instance, a for loop statement containing another for loop. The syntax of Java does not set a limit on the number of levels of nesting.

3.7 Arrays

Arrays are data structures that hold data of the same type in contiguous memory. A group of memory cells that are contiguous have the same name and store the *same* data type. The purpose of an array is to store data where each piece of data is of the same type.

3.7.1 One-dimensional arrays

3.7.1.1 Declaration of one-dimensional arrays

All arrays must be declared before they can be used. The general way to declare a one-dimensional array is to declare the *type* of the array, then the *name* of the array followed by the brackets []. This is done as in the following examples:

```
int age[ ];
double score[ ];
String lastnames[ ];
```

In Java, arrays can be declared anywhere, just like any other variable. Failure to declare an array will result in a compile-time error message.

3.7.1.2 Allocation of memory and initialization

Memory must be allocated for an array before it is used. This is done by making a statement as given in the following examples. In the statement, the data type is mentioned again and this must be the same as the data type that was indicated when the array was declared.

```
age = new int[10];
score = new double[100];
lastnames = new String[1000];
```

If array size is declared as N then the elements are numbered from 0 ... (N–1).

In the example given, age has elements 0 ... 9, score has elements 0 ... 99, and lastnames has elements 0 ... 9999.

An array can be declared and created in a single statement.

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

Accessing an element that does not exist (or is out of scope) causes a run-time error and the program terminates abruptly at that point. Such an error is indicated by the phrase Array out of Index.

Normally, in Java, memory is allocated for arrays in the constructor for application programs and in the init() function for applets.

3.7.1.3 Using arrays

After an array is declared, elements within the array are accessed by providing an index (that is placed in brackets) The value inside the square brackets [ ] indicates the index corresponding to the element that is to be accessed. Thus, the value**[i]** accesses the *i*th element of the array. The array index can be a literal value, a variable or an expression.

Array indices in Java begin at 0 and end at the array length minus 1. The length property is referenced to obtain the size of an array (number of elements). A value is assigned to an array element through the combined usage of the array name and the index. This assignment can be done as shown in the following examples:

```
area[0] = 25;
total[i] = 100;
name[t–1] = "Ravi"
```

Accessing a value in an array element is similar to assigning a value to an element in an array. This is done as illustrated in the following examples:

```
varAge = age[0];
varScore= score[i]
```

Program 3.23 creates an array, inserts some values in it and displays those values.

Program 3.23 Printing an array of integers.

```
public class DisplayArray
{
    public static void main(String[ ] args)
    {
        int[] myArray = new int[10];  // declare and create an array of integers
            // assign a value to each array element and print
        for (int i = 0; i < myArray.length; i++)
        {
            myArray[i] = i;
            System.out.print(myArray[i] + " ");
        }
        System.out.println();
    }
}
```

3.7.2 Multi-dimensional arrays

Multi-dimensional arrays (or multiple-subscripted arrays) can be used to represent tables of data. A two-dimensional array has rows and columns and two subscripts are needed to access the array

elements. Java allows creation of a two-dimensional array containing n rows and m columns (that is, an $n \times m$ array).

Java represents multi-dimensional arrays using an one-dimensional array whose individual elements are once again members of an one-dimensional array. Using this definition we can create arrays that are two-dimensional or more than that. Usually the highest dimension that is used is three, beyond which visualizing an array is cumbersome.

Elements of multi-dimensional arrays are accessed in a way similar to that of an one-dimensional array. Instead of using a single subscript, the n-dimensional array uses n subscripts. A two-dimensional array uses two subscripts and so on. A two-dimensional array can occur in a statement in the following manner:

```
arrayName[i][j] = value;
```

The declaration of multi-dimensional arrays is similar to one-dimensional arrays. This is done by a statement such as the following one:

```
int arrayOfNums [ ][ ];
```

The number of columns in each row of a multi-dimensional array should be same. This must be observed while defining it.

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

Initializing values in a multi-dimensional array is also similar to initialization in one-dimensional arrays. For the array arrayOfNums, this is done as shown below:

```
int[ ][ ] arrayOfNums = {{13, 23, 33, 44}, {14, 24, 34, 44}, {15, 25, 35, 45}};
```

Program 3.24 creates a two-dimensional array, inserts some values in it and displays the values.

Program 3.24 Using two-dimensional arrays.

```
public static void main (String args[ ])
{
    int[ ][ ] matrix;
    matrix = new int[4][5];
    for (int row=0; row < 4; row++)
    {
        for (int col=0; col < 5; col++)
```

```
        {
            matrix[row][col] = row+col;
        }
    }
    for (int row=0; row < 4; row++)
    {
        for (int col=0; col < 5; col++)
        {
            System.out.print( matrix[row][col]+ " ");
        }
        System.out.println("");
    }
}
```

The output of Program 3.24 is the following:

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

3.8 Strings

A string represents a group of characters. Strings written in a Java program are all String class objects. String class objects are immutable (cannot be modified) because the time taken by JVM to reallocate memory and add data to an existing object is more than the time taken to create a new object. A new object is created every time a modification is required. Java defines the String class, which is part of the java.lang package. The languages such as C, C++ implement strings as Character array whereas Character array and strings are different in Java. This class allows programmers to perform operations such as concatenation, sub-string and case changes (Lower–Upper case and Upper–Lower).

3.8.1 Constructors of String class with character array and String object

Constructors have been mentioned earlier and will be explained in detail in chapter 4. They can take any of the following forms:

```
String (char array[ ]);
String (char array[ ], int start , int count);
String(String object);
```

3.8.2 Constructors of String class with byte array

These type of constructors can take the following forms:

```
String(byte a[ ]);
String(byte a[ ],int start, int count);
String(byte a[ ], int start, int count, String encodingchars);
```

The String class also defines a special type of constructor that contains StringBuffer parameter.

```
String(StringBuffer a[ ]);
```

Program 3.25 creates Strings using constructors.

Program 3.25 Using constructors to create strings.

```
import java.io.*;
class SampleString
{
    public static void main(String[ ] args)
    {
        String str1=new String("Program");
        byte s1[ ]= new byte[5];
        for(int i=0;i<5;i++)
            s1[i]=(byte)(i+65);
        String str2=new String(s1);
        String str3=new String(s1,2,2);
        char c[ ]={'j','a','v','a'};
        String str4 = new String(c);
        String str5 = new String(c,2,2);
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
        System.out.println(str4);
        System.out.println(str5);
    }
}
```

Output:

```
Program
ABCDE
```

CD
java
va

3.8.3 Methods of the String class

In all the methods of the String object, the index of the characters starts from 0 to length()–1.

Table 3.14 lists various methods in the String class

Table 3.14 Methods in the String class.

Method	Description
char charAt(int index)	Returns the character at the specified index. Otherwise it returns StringIndexOutOfBoundException.
void getChars(int sourcebegin, intsourceend, char[ ] destination, int destinationbegin)	This method will copy (sourceend–sourcebegin) –1 number of characters starting from the destinationbegin. In the case of errors it will throw the StringIndexOutOfBoundException and the NullPointerException.
Byte[ ] getBytes( )	Returns the byte array of the specified string.
boolean equals(String str)	Returns true if the String object specified using str is equal to the invoking string. Otherwise, returns the value false. This is a case-sensitive method.
boolean equalsIgnoreCase(String str)	This method will do the same thing as the equals() method except that it will ignore the case.
int compareTo(String str)	It will return 1 if the String object str is less than the invoking String, return –1 if str is greater than the invoking String and returns 0 if both are equal.
int compareToIgnoreCase(String str)	This method will do the same as compareTo( ) except that it will ignore the case.
boolean startsWith(String str) boolean startsWith(String str, int index)	It will return true if the invoking string starts with str. The second version of this method will compare the invoking string with str from index which is specified.
boolean endsWith(String str) boolean endsWith(String str, int index)	It will return true, if the invoking string ends with string str. The second form of this method will compare str from the index specified.
int indexOf(int ch) int indexOf(int ch, int index)	It will return the index of the first occurrence of character specified by ch. In the second form index will specify the start index and goes through the end of the string.

(*Table* 3.14 *Contd.*)

(*Table 3.14 Continued*)

Method	Description
int lastIndexOf(int ch) int lastIndexOf(int ch, int index)	It will return the last occurrence of the character specified by ch. It starts searching backward from the specified index and returns the index within the string of the last occurence.
String subString(int index) String subString(int index1, int index2)	It will return the sub-string of the string on which it is invoked starting from the index. In the second form of the subString() method index1 specifies the starting index and index2 specifies the end index.
String concat(String str)	This method will return the string in which str is concatenated with the invoking string.
String replace(char oldcharacter, char newcharacter)	It will return the string with oldcharacter replaced by newcharacter.
String trim()	It will return the string by removing the left-most and right-most white spaces.
String toLowerCase( )	Returns the string which contains only lowercase letters. That is, all letters in capitals are converted to lowercase letters.
String toUpperCase( )	Returns the string which contains only capital letters. That is, all the lowercase letters are converted into capital letters.
int length( )	Returns the length of the string on which it is invoked.

3.8.3.1 The valueOf method

In general, concatenation (+) operations are performed on all data types with String data type. First the valueOf( ) method is called, with that data type value as parameter. This will return its String value. Then this String value is appended to the other String. What happens during append is seen in the section on StringBuffer. This method is a static method which is overloaded.

```
static String valueOf(double value)
static String valueOf(long value)
static String valueOf(int value)
static String valueOf(float value)
static String valueOf(boolean value)
static String valueOf(char[ ] value)
static String valueOf(char[ ] value, int start, int length) //It converts character array
                                                            //starting from the 'start' to the 'length' specified.
static String valueOf(Object value)
```

3.8.3.2 The equals() method

Program 3.26 illustrates the use of the equals() method.

Program 3.26 Using the equals() method to compare Strings.

```
class TestEquals
{
    public static void main(String args[ ])
    {
        String str1= new String("java");
        String str2=new String("java");
        String str3=new String(str1);
        if(str1.equals(str2))
            System.out.println("str1.equals(str2)is true");
        else
            System.out.println("str1.equals(str2)is false");
        if(str1.equals(str3))
            System.out.println("str1.equals(str3) is true");
        else
            System.out.println("str1.equals(str3) is false");
        if(str2.equals(str3))
            System.out.println("str2.equals(str3) is true");
        else
            System.out.println("false");
        if (str1 == str2)
            System.out.println("str1==str2 is true");
        else
            System.out.println("str1==str2 is false");
        if(str1 == str3)
            System.out.println("str1==str3 is true");
        else
            System.out.println("str1==str3 is false");
        if(str2 == str3)
            System.out.println("str1==str3 is true");
        else
            System.out.println("str1==str3 is false");
    }
}
```

Output:

```
str1.equals(str2) is true
str1.equals(str3) is true
str2.equals(str3) is true
str1= = str2 is false
str1= = str3 is false
str2= =str3 is false
```

The equals() method compares the contents of the two strings whereas the '= =' compares the object references of the string objects.

3.9 The StringBuffer Class

An object of the String class is immutable. In order to achieve mutability, Java provides a class called StringBuffer. A StringBuffer is a mutable string, which can be changed during the program. It is a changeable set of characters that can grow in size. When ever the + (concatenation) operator is used on string objects, the string will be changed into an instance of StringBuffer and the append() method is called on that instance of StringBuffer. After that, StringBuffer is changed into String by using the toString() method of the Object class. StringBuffer also facilitates the insertion of a character in the middle of a string.

3.9.1 Constructors of the StringBuffer class

There is a default constructor that creates a StringBuffer with a capacity of 16 characters. This takes the form shown below:

```
StringBuffer();
```

The following type of constructor will specify the size of the StringBuffer.

```
StringBuffer(int size);
```

The following type of the constructor will create the StringBuffer with the String object.

```
StringBuffer(String str);
```

Whenever the size of the StringBuffer is not specified, it will allocate space for the 16 additional characters because reallocation is a costly process. These are some methods which are not part of the String object. StringBuffer also supports all the methods that are supported by String.

Table 3.15 lists various methods in StringBuffer class.

Table 3.15 Methods belonging to the StringBuffer class.

Method	Description
StringBuffer append(int value) StringBuffer append(char[] str, int start, int length)	This method will return StringBuffer after appending value to its current form. This method holds good for all data types, String class and Object class The second form will append the character array at the position start to the length of characters to StringBuffer.
int capacity()	It will return the total capacity of the StringBuffer.
StringBuffer delete(int start, int end)	It will return the StringBuffer after removing the characters from the index start to the index (end–1).
StringBuffer deleteCharAt(int index)	Returns the StringBuffer by removing the character at the index.
StringBuffer insert(int index, char c) StringBuffer insert(int index, char[] str, int start, int length)	This method will insert the character at the index specified and returns the StringBuffer. This method also holds for all data types, String and Object. The second form of this method will insert the sub-array starting from the index start up to the point specified by length at the point index of the StringBuffer.
StringBuffer reverse( )	Returns the StringBuffer which is reversed in order compared with the original one.
void setCharAt(int index, char ch)	It will replace the character at index by the character ch in the StringBuffer.
Void setLength(int size)	It will set the length of the StringBuffer to size.

Objective type questions

1. The value of the following expression is
   int k=5 | 2 is _____.
2. In a class global variables are declared using the keyword _____.
3. _____ is used to represent a back slash(\) character.
4. Which of the following is an illegal statement in java.
   a. int i=–2; b. float f=45.0; c. double d=33.6; d. char a=17;
5. The value of 128>>>1 is
   a. 63. b. –63. c. –128. d. 32.