Unit 3 Operators and Control Statements

Structure:

3.1 Introduction

In the last unit, you have learnt the basic program structure in Java. In this unit, we will explore various operators and control statements in Java. There are many constructs in Java to implement **c**onditional execution. For example flow of a program can be controlled using the ***if*** construct. The ***if*** construct allows selective execution of statements depending on the value of the expressions associated with the ***if*** construct. Similarly repetitive tasks can be done using ***for*** constructs which would be discussed in detail in this unit.

Objectives:

After studying this unit, you should be able to:

- explain different operators in Java
- discuss various control statements in Java

3.2 Operators

Operators play an important role in Java. There are three kinds of operators in Java. They are (i) Arithmetic Operators (ii) Comparison / Relational Operators and (iii) Logical Operators

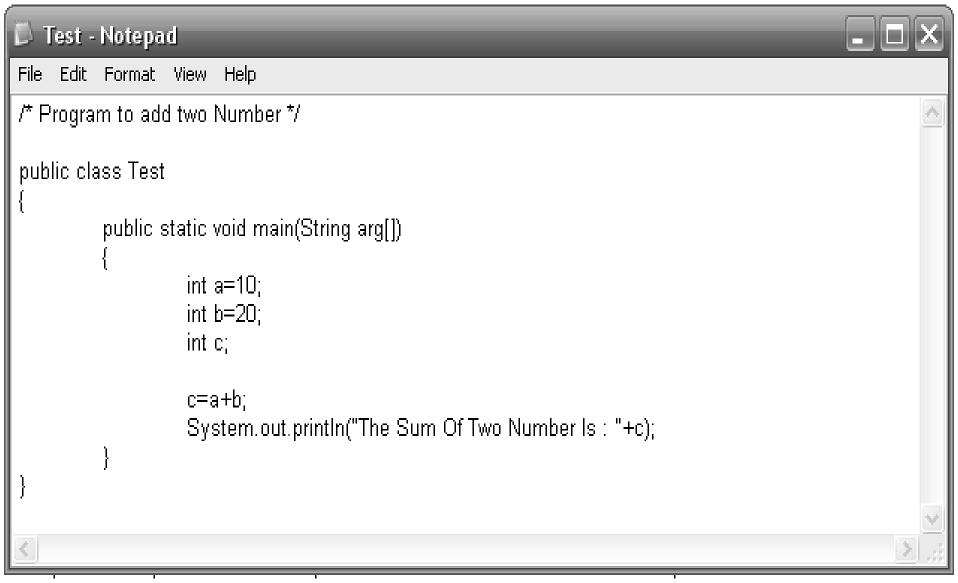
3.2.1 Arithmetic Operators

Addition, Subtraction, Multiplication, Division and Modulus are the various arithmetic operations that can be performed in Java.

Table 3.1: List of Arithmetic Operators

Operator	**Meaning**	**Use**	**Meaning**
+	Addition	op1+op2	Adds op1 and op2
-	Subtraction	op1-op2	Subtracts op2 from op1
*	Multiplication	op1*op2	Multiplies op1 and op2
/	Division	op1/op2	Divides op1 by op2
%	Modulus	op1 % op2	Computes the remainder of dividing op1 by op2

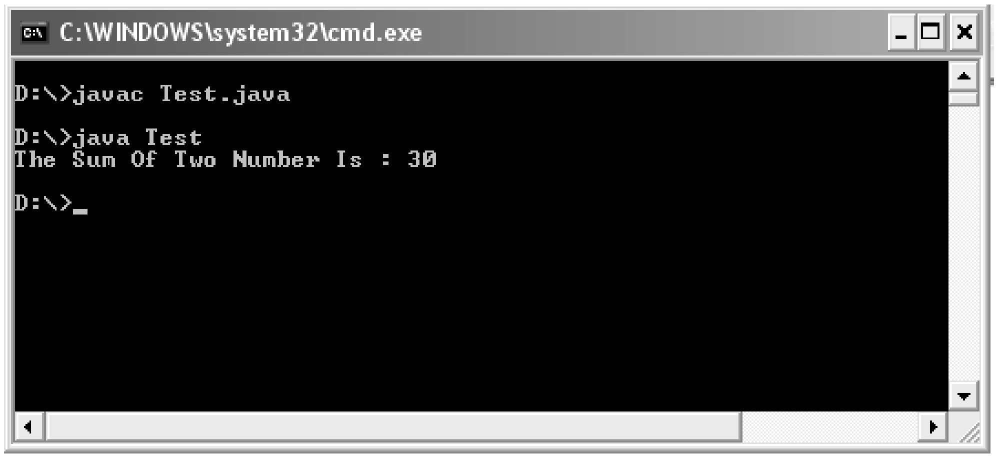
The following Java program adds two numbers and prints the result.



```
/* Program to add two Number */

public class Test
{
        public static void main(String arg[])
        {
                int a=10;
                int b=20;
                int c;

                c=a+b;
                System.out.println("The Sum Of Two Number Is : "+c);
        }
}
```

Figure 3.1: Java Program to add two numbers and printing the result

The compilation and running of the program is shown in figure 3.2.

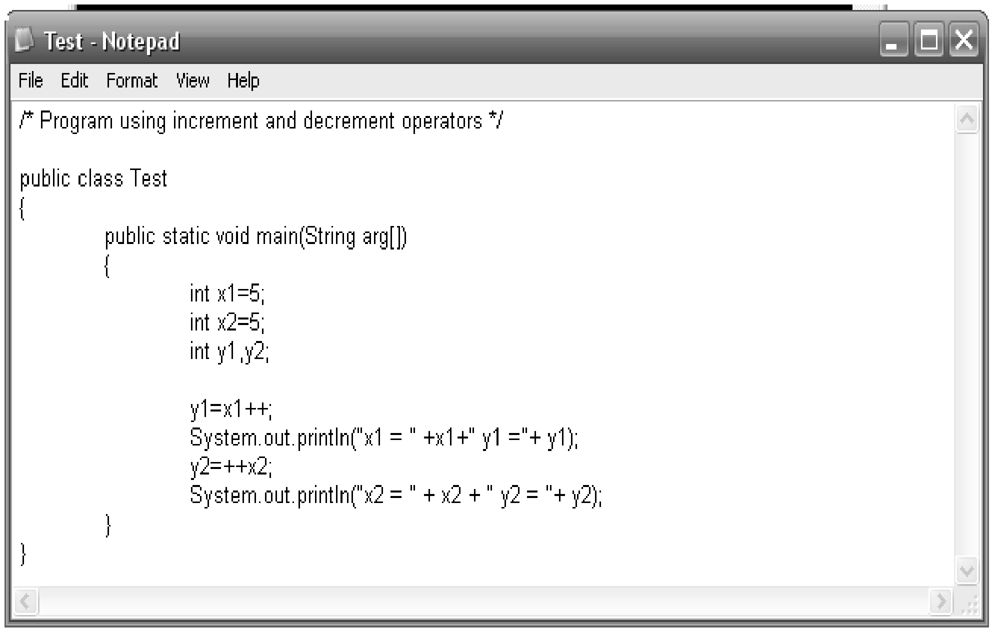


```
D:\>javac Test.java

D:\>java Test
The Sum Of Two Number Is : 30

D:\>_
```

Figure 3.2: Compilation and Running of Java Program

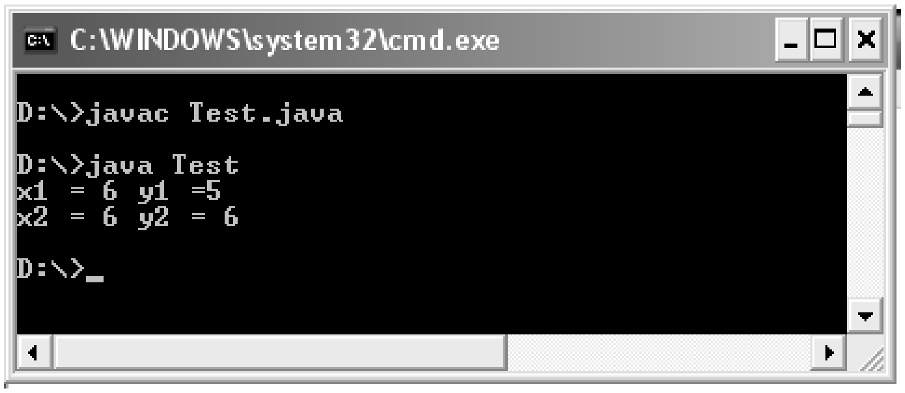
3.2.2 Increment and Decrement Operators

The increment operator is **++** and decrement operator is **--**. This is used to add 1 to the value of a variable or subtract 1 from the value of a variable. These operators are placed either before the variable or after the variable name. The example below shows the use of these operators.



```
/* Program using increment and decrement operators */

public class Test
{
        public static void main(String arg[])
        {
                int x1=5;
                int x2=5;
                int y1,y2;

                y1=x1++;
                System.out.println("x1 = " +x1+" y1 ="+ y1);
                y2=++x2;
                System.out.println("x2 = " + x2 + " y2 = "+ y2);
        }
}
```

Figure 3.3: Example showing increment operators in Java



```
D:\>javac Test.java

D:\>java Test
x1 = 6 y1 =5
x2 = 6 y2 = 6

D:\>_
```

Figure 3.4: Program Compilation and Running

When the operator **++** is placed after the variable name, first the assignment of the value of the variable takes place and then the value of the variable is incremented. This operation is also called ***post increment*.** Therefore the value of y1 will remain as 5 and the value of x1 will be 6. When the operator is placed before the variable, first increment of the variable takes place and then the assignment occurs. Hence the value x2 and y2 both will be 6. This operation is also called as ***pre increment*.** Similarly **– –** operator can be used to perform ***post decrement*** and ***pre decrement*** operations. If there is no assignment and only the value of variable has to be incremented or decremented then placing the operator after or before does not make difference.

3.2.3 Comparison Operators

Comparison operators are used to compare two values and give the results.

Table 3.2: List of Comparison Operators in Java

Operator	**Meaning**	**Example**	**Remarks**
= =	Equal	op1 = = op2	Checks if op1 is equal to op2
!=	Not Equal	op1 != op2	Checks if op1 is not equal to op2
<	Less than	op1 < op2	Checks if op1 is less than op2
>	Greater than	op1 > op2	Checks if op1 is greater than op2
<=	Less than or equal	op1 <= op2	Checks if op1 is less than or equal to op2
>=	Greater than or equal	op1 >= op2	Checks if op1 is greater than or equal to op2

3.2.4 Logical Operators

Logical operators are used to perform Boolean operations on the operands.

Table 3.3: List of Logical Operators in Java

Operator	**Meaning**	**Example**	**Remarks**
&&	Short-circuit AND	op1 && op2	Returns true if both are true. If op1 is false, op2 will not be evaluated and returns false.
||	Short-circuit OR	op1 || op2	Returns true if anyone is true. If op1 is true, op2 will not be evaluated and returns true.
!	Logical unary NOT	!op	Returns true if op is false.
&	Logical AND	Op1 & op2	Returns true if both are true. Always op1 and op2 will be evaluated.
|	Logical OR	Op1 | op2	Returns true if anyone is true. Always op1 and op2 will be evaluated.

3.2.5 Operator Precedence

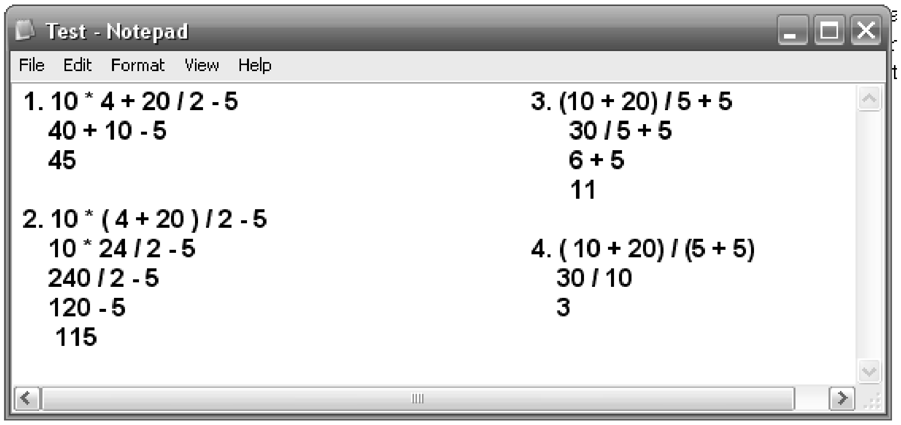
When more than one operator is used in an expression, Java will use operator precedence rule to determine the order in which the operators will be evaluated. For example, consider the following expression:
Result=10+5*8-15/5

In the above expression, multiplication and division operations have higher priority over the addition and subtraction. Hence they are performed first. Now, Result = 10+40-3.

Addition and subtraction has the same priority. When the operators are having the same priority, they are evaluated from left to right in the order they appear in the expression. Hence the value of the result will become 47. In general the following priority order is followed when evaluating an expression:

- Increment and decrement operations.
- Arithmetic operations.
- Comparisons.
- Logical operations.
- Assignment operations.

To change the order in which expressions are evaluated, parentheses are placed around the expressions that are to be evaluated first. When the parentheses are nested together, the expressions in the innermost parentheses are evaluated first. Parentheses also improve the readability of the expressions. When the operator precedence is not clear, parentheses can be used to avoid any confusion.



```
1. 10 * 4 + 20 / 2 - 5              3. (10 + 20) / 5 + 5
   40 + 10 - 5                          30 / 5 + 5
   45                                   6 + 5
                                        11
2. 10 * ( 4 + 20 ) / 2 - 5
   10 * 24 / 2 - 5                  4. ( 10 + 20) / (5 + 5)
   240 / 2 - 5                         30 / 10
   120 - 5                             3
    115
```

Figure 3.5: Operator Precedence Example

Self Assessment Questions:

1. Give the symbol for modulus operator.
2. Give the symbol for logical AND operator.

3.3 Control Flow Statements

Following statements are used to control the flow of execution in a program:

1. Decision Making Statements
   - If-else statement
   - Switch – case statement
2. Looping Statements
   - For loop
   - While loop
   - Do-while loop
3. Other statements
   - Break
   - Continue

3.3.1 If-else statement

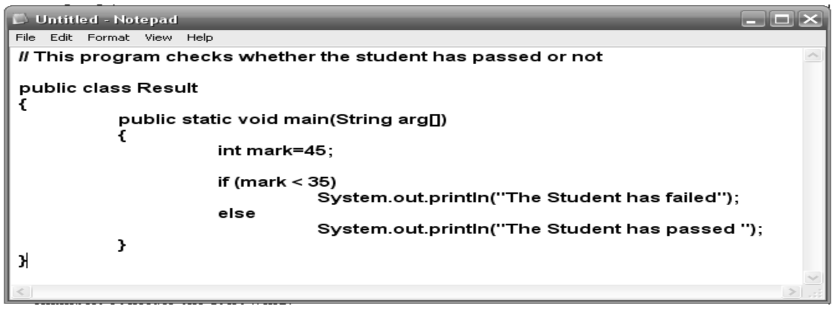
The ***if*** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general syntax of the ***if*** statement:

if (condition) statement1;

else statement2;

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

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



```
// This program checks whether the student has passed or not

public class Result
{
            public static void main(String arg[])
            {
                        int mark=45;

                        if (mark < 35)
                                    System.out.println("The Student has failed");
                        else
                                    System.out.println("The Student has passed ");
            }
}
```

Figure 3.6: If… else statement example

Most often, the expression used to control the ***if*** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment:

```
boolean dataAvailable;
// ...
if (dataAvailable)
ProcessData();
else
waitForMoreData();
```

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you'll need to create a block, as in this fragment:

```
int bytesAvailable;
// ...
```

```
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
```

Here, both statements within the **if** block will execute if **bytesAvailable** is greater than zero. Some programmers find it convenient to include the curly braces when using the **if**, even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don't have to worry about forgetting the brackets. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
bytesAvailable = n;
```

It seems clear that the statement **bytesAvailable = n;** was intended to be executed inside the **else** clause, because of the indentation level. However, as you recall, whitespace is insignificant to Java, and there is no way for the compiler to know what was intended. This code will compile without complaint, but it will behave incorrectly when run.

The preceding example is fixed in the code that follows:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else {
```

```
waitForMoreData();
bytesAvailable = n;
}
```

The if-else-if Ladder

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

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

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

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

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

Here is the output produced by the program:

```
April is in the Spring.
```

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

3.3.2 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.

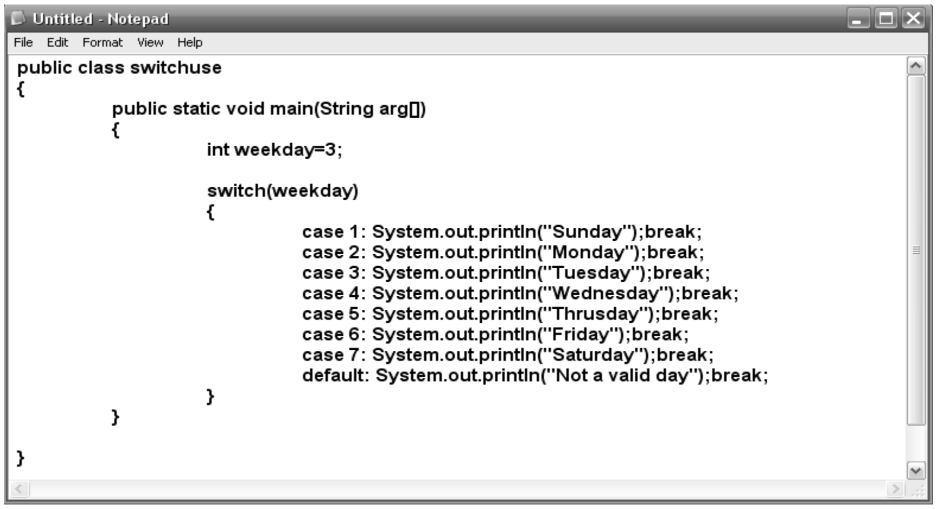
Here is the general form of a **switch** statement:

```
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. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The **switch** statement works as follows: 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**.



```
public class switchuse
{
        public static void main(String arg[])
        {
                int weekday=3;

                switch(weekday)
                {
                        case 1: System.out.println("Sunday");break;
                        case 2: System.out.println("Monday");break;
                        case 3: System.out.println("Tuesday");break;
                        case 4: System.out.println("Wednesday");break;
                        case 5: System.out.println("Thrusday");break;
                        case 6: System.out.println("Friday");break;
                        case 7: System.out.println("Saturday");break;
                        default: System.out.println("Not a valid day");break;
                }
        }

}
```

Figure 3.7: The switch…case statement example

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
public static void main(String args[ ]) {
for(int i=0; i<12; i++)
switch(i) {
case 0:
case 1:
case 2:
case 3:
case 4:
System.out.println("i is less than 5");
break;
case 5:
case 6:
case 7:
case 8:
case 9:
System.out.println("i is less than 10");
break;
default:
System.out.println("i is 10 or more");
}
}
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
```

```
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

Nested switch Statements

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

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

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

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

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of **Boolean** expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement enclosed by an 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**. 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.

3.3.3 For Loop

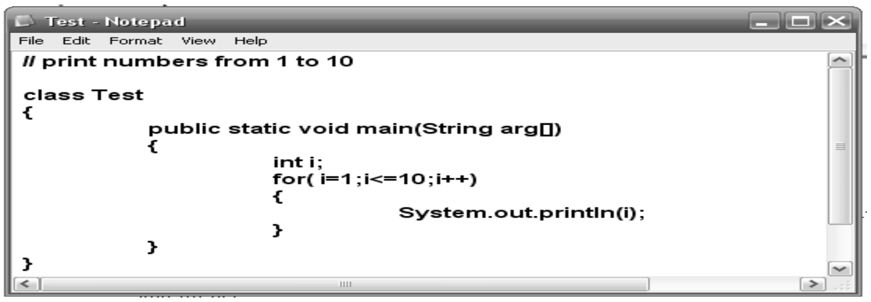
The usage of **for** loop is as follows:

```
for (initial statement; termination condition; increment instruction)
statement;
```

When multiple statements are to be included in the **for** loop, the statements are included inside flower braces as below:

```
for (initial statement; termination condition; increment instruction)
{
        Statement 1;
        statement 2;
}
```

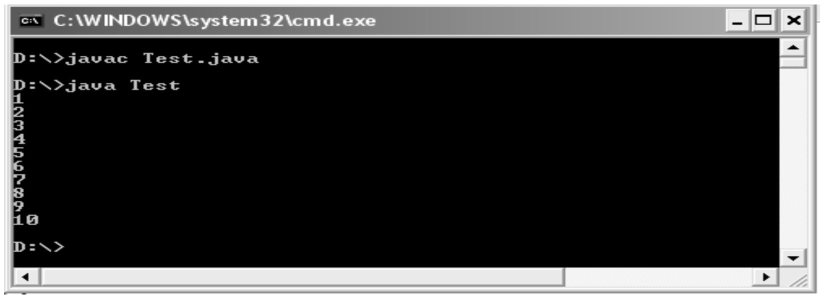
The example below prints numbers from 1 to 10.



```
// print numbers from 1 to 10

class Test
{
		public static void main(String arg[])
		{
				int i;
				for( i=1;i<=10;i++)
				{
						System.out.println(i);
				}
		}
}
```

Figure 3.8: For loop example program

The result of the above program is shown below:



```
D:\>javac Test.java

D:\>java Test
1
2
3
4
5
6
7
8
9
10

D:\>
```

Figure 3.9: Result of the above for loop program

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

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

The output produced by this program is shown here:

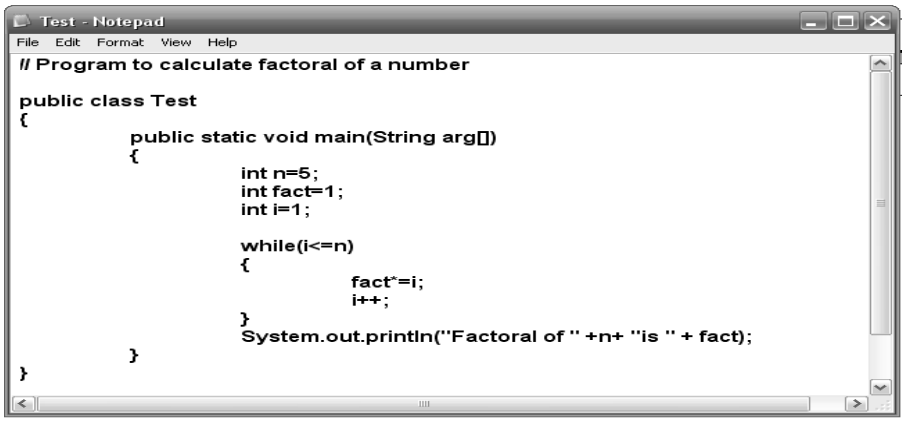
```
..........
.........
........
.......
......
.....
```

3.3.4 While Loop

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

> **while (*condition*) {**
> **// body of loop**
> **}**

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



```
// Program to calculate factoral of a number

public class Test
{
            public static void main(String arg[])
            {
                        int n=5;
                        int fact=1;
                        int i=1;

                        while(i<=n)
                        {
                                    fact*=i;
                                    i++;
                        }
                        System.out.println("Factoral of " +n+ "is " + fact);
            }
}
```

Figure 3.10: While loop example

3.3.5 Do…. While Loop

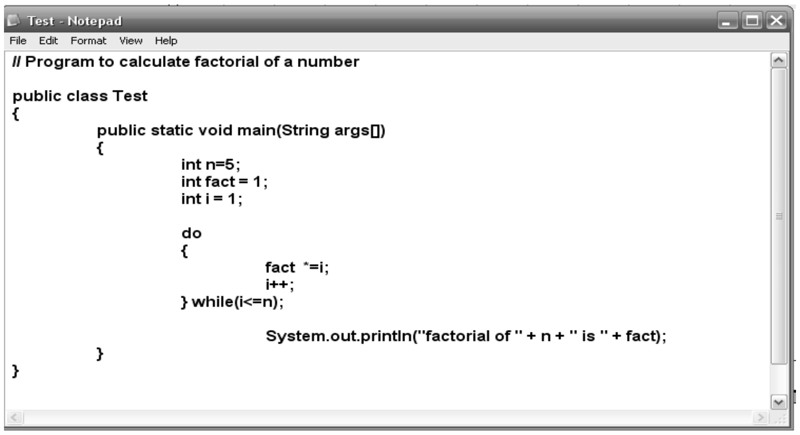
As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In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

do {

// body of loop

} while (*condition*);

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



```
// Program to calculate factorial of a number

public class Test
{
            public static void main(String args[])
            {
                        int n=5;
                        int fact = 1;
                        int i = 1;

                        do
                        {
                                    fact  *=i;
                                    i++;
                        } while(i<=n);

                                    System.out.println("factorial of " + n + " is " + fact);
            }
}
```

Figure 3.11: Do…while loop example

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program which implements a very simple help system for Java's selection and iteration statements:

```
// Using a do-while to process a menu selection
class Menu {
public static void main(String args[])
throws java.io.IOException {
```

```
char choice;
do {
System.out.println("Help on:");
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.println(" 3. while");
System.out.println(" 4. do-while");
System.out.println(" 5. for\\n");
System.out.println("Choose one:");
choice = (char) System.in.read();
} while( choice < '1' || choice > '5');
System.out.println("\\n");
switch(choice) {
case '1':
System.out.println("The if:\\n");
System.out.println("if(condition) statement;");
System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\\n");
System.out.println("switch(expression) {");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" // ...");
System.out.println("}");
break;
case '3':
System.out.println("The while:\\n");
System.out.println("while(condition) statement;");
```

```
        break;
        case '4':
        System.out.println("The do-while:\\n");
        System.out.println("do {");
        System.out.println(" statement;");
        System.out.println("} while (condition);");
        break;
        case '5':
        System.out.println("The for:\\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break; } } }
```

Here is a sample run produced by this program:

```
Help on:
1. if
2. switch
3. while
4. do-while
5. for
Choose one:
The do-while:
do {
statement;
} while (condition);
```

In the program, the **do-while** loop is used to verify that the user has entered a valid choice. If not, then the user is re-prompted. As the menu must be displayed at least once, the **do-while** is the perfect loop to accomplish this.

A few other points about this example: Notice that the characters are read from the keyboard by calling **System.in.read( )**. This is one of Java's console input functions. Although Java's console I/O methods won't be discussed in detail, just know that **System.in.read( )** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program.

Java's console input is quite limited and awkward to work with. Further, most real-world Java programs and applets will be graphical and window-based. For these reasons, not much use of console input has been made in this book. However, it is useful in this context. One other point; because **System.in.read( )** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors.

3.3.6 Break Statement

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

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

This program generates the following output:

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

```
i: 9
Loop complete.
```

As you can see, although the **for** loop is designed to run from 0 to 99, the **break** statement causes it to terminate early, when **i** equal 10.

3.3.7 Continue Statement

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

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

```
// Demonstrate continue.
class Continue {
public static void main (String args[ ]) {
for (int i=0; i<10; i++) {
System.out.print (i + " ");
if (i%2 == 0) continue;
System.out.println ("");
}
}
}
```

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

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

As with the **break** statement, **continue** may specify a label to describe which enclosing loops to continue.

Self Assessment Questions:

3. If-else is a looping statement. (True or False)
4. Do…While is a decision making statement. (True or False)

3.4 Summary

This unit has given a brief overview of various operators and conditional statements in Java. Let us summarize the concepts:

- **Implementing Conditional Execution**
  You can control the flow of a program using the **if** construct. The **if** construct allows selective execution of statements depending on the value of the expressions associated with the if construct.
- **Performing Repetitive Tasks Using Loop Construct**
  You can perform repetitive tasks by making use of the **for** construct.

3.5 Terminal Questions

1. What are the different types of operators used in Java?
2. What do you understand by operator precedence?
3. What are the different types of control statements?
4. Write Java program to print the address of the study center.
5. Write Java program to convert the Rupees to Dollars.
6. Write a Java program to compare whether your height is equal to your friends height.
7. Write a Java program to find the sum of 1+3+5+…. for 10 terms in the series.

3.6 Answers

Self Assessment Questions

1. %
2. &
3. False
4. False

Terminal Questions

1. Arithmetic, Comparison and Logical Operators. (Refer Section 3.2.1 to 3.2.4)

2. When more than one operator is used in an expression, Java will use operator precedence rule to determine the order in which the operators will be evaluated. (Refer Section 3.2)
3. If…else, Switch…Case, For, While, Do…While, Break, and Continue. (Refer Section 3.3.1 to 3.3.7)

(Terminal Questions 4 to 7 are programming exercises. For this, you should go through this unit thoroughly.)