UNIT 4 EXPRESSIONS, STATEMENTS AND ARRAYS

Structure **Page Nos.**

4.0 INTRODUCTION

Variables and operators, which you studied in the last unit, are basic building blocks of programs. Expressions are segments of code that perform some computations and return some values. Expressions are formed by combining literals, variables and operators. Certain expressions can be made into statements, which are complete units of execution. Statements are normally executed sequentially in the order in which they appear in the program. But there are number of situations where you may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified condition(s) are met. Java language supports various control statements that can be put into the following categories: selection, iteration and jump. You will learn use of these in Java programming. An *array* is a data structure supported by all the programming languages that stores multiple values of the same type in a fixed length structure.

In this unit first we will cover about expressions and their evaluation. We will discuss about *operator precedence*. Then we will cover various kinds of statements supported in Java. At the end we will discuss arrays in Java language.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- define expression;
- define and write the various kinds of statements;
- write programs using sequential, conditional, and iterative statements, and
- use array in programming.

4.2 EXPRESSIONS

Arithmetic expressions consist of operators, operands, parentheses, and function calls.

You studied in your school mathematics about mathematical expressions, made up of simpler terms or variables. Similarly in Java, expression is built from simpler expressions or variables using the operators. In this section we look at the various operators and show how they can be combined to generate expressions. **Expressions** perform the work of a program. They are used to compute and to assign values to

variables. Also, they help in controlling the execution flow of a program. The job of an expression is to perform the computation as indicated by the elements of the expression. Finally, it returns a value that is the result of the computation.

Definition: An *Arithmetic expression* is a series of variables, operators, and method calls that evaluates to a single value. It is constructed according to the syntax of the language.

As discussed in the previous unit, the application of operators returns a value, so that you can say the expression is nothing but a use of operators. For example if you write code "index++;" then '++' is an operator used with variable "index" and "index++;" is an expression.

The program given below shows some of the expressions given in **bold**:

```
…….
// primitive types
char Respond = 'Y';
boolean a Boolean = true;

// display all
System.out.println ("The largest byte value is " + largest Byte);
...

if (Character.isUpperCase (Respond))
{
   …….
}
```

Each of these expressions performs an operation and returns a value. As you can see, it is explained in *Table 1*.

Table 1: Arithmetic expression

Expression	Action	Value Returned
Respond = 'Y'	Assign the character 'Y' to the character variable Respond	The value of Respond after the assignment is ('Y')
"The largest byte value is " + largest Byte	Concatenate the string "The largest byte value is " and the value of largest Byte converted to a string	The resulting string: The largest byte value is 127
Character.isUpperCase (Respond)	Call the method is Upper Case	The return value of the method: true

In an expression, the data type of the value returned depends on the elements used. The expression **Respond = 'Y'** returns a character. Since both the operands Respond and 'Y' are characters, the use of assignment operator returns a character value. As you observe from the other expressions, an expression can return a boolean, a string, and so on.

The Java programming language allows programmers to construct **compound expressions** and statements from smaller expressions. It is allowed as long as the data types required by one part of the expression matches the data types of the other. Let us consider the following example of a compound expression:

x * y / z

In this example, the order in which the expression is evaluated is unimportant because the results of multiplication and division are *independent of order*. However, it is not true for all expressions. Let us consider another *example*, where addition and division operations are involved. Different results will be there depending on the *order of operation* i.e. whether you perform the addition or the division operation first:

```
x + y / 50     //ambiguous expression
```

The programmer can specify exactly how s/he wants an expression to be evaluated by using balanced parentheses **'('** and **')'**. For example s/he could write:

```
(x + y)/ 50   //unambiguous and recommended way
```

If the programmer doesn't explicitly indicate the order of the operations in a compound expression to be performed, it is determined by the **precedence** assigned to the operators. Operators with a higher precedence get evaluated first. Let us consider the example again; the division operator has a higher precedence than the addition operator. Thus, the two following statements are equivalent:

```
x + y / 50
x + (y / 50)
```

Therefore, it is suggested that while writing compound expressions, you should be explicit by using parentheses to specify which operators should be evaluated first. This will make the code easier to read and maintain.

Table 2 shows the operator precedence. The operators in this table are listed in order of the following precedence rule: the higher in the table an operator appears, the higher its precedence. In an expression, operators that have higher precedence are evaluated before operators that have relatively lower precedence. Operators on the same line have equal precedence.

Table 2: Operators precedence

Special operators	[ ] Array element reference Member selection (*params*) Function call
unary operators	++, --, +, -, ~, !
Creation or cast	new (*type*)*expression*
multiplicative	*, /, %
Additive	+, -
Shift	<<, >>, >>>
Relational	<, >, <=, >=, instance of
Equality	= =, !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|

logical AND	&&
logical OR	| |
Conditional	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

When operators of same precedence appear in the same expression, some rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated in left-to-right order and assignment operators are evaluated right to left.

4.3 STATEMENTS

Statements in Java are equivalent to sentences in natural languages. A *statement* is a complete unit of execution. It is an executable combination of tokens ending with a semicolon (;) mark. A token can be any variable, constant, operator or an expression. By terminating the expression with a semicolon (;), the following types of expressions can be made into a statement

- Assignment expressions
- Any use of ++ or --
- Method calls
- Object creation expressions

These kinds of statements are called *expression statements*. Let us consider some examples of expression statements:

```
piValue = 3.141;                          //assignment statement
counter++;                                //increment statement

System.out.println(piValue);              //method call statement

Integer integerObject = new Integer(4);   //object creation statement
```

In addition to these kinds of expression statements, there are two other kinds of statements.

- A *declaration statement* declares a variable. Let us consider the following examples of declaration statements.
  ```
  Integer counter;
  double xValue = 91.224;
  ```
- A *control statement* regulates the flow of execution of statements based on the changes to the state of a program.

The while loop, **for** loop and the **if** statement are examples of control flow statements. This category of statements is considered in depth in coming sections.

Blocks

A *code **block*** is a group of two or more statements between balanced braces. It is a single logical unit that can be used anywhere. Even a single statement is allowed in this logical unit. The following listing shows a block that is target for if statement:

```
if (x < y)
{   //beginning of block
```

```
    System.out.println("The value of x is less than y" );
    x = 0;
}  // Ending of block
```

In this example, if x is less than y, both statements inside the block will be executed. Thus, both the statements inside the block form a logical unit. The key point here is that whenever you need to logically link two or more statements, you do so by creating a code block.

☞ Check Your Progress 1

1) What are the data types of the following expressions? Assuming that "i" is an integer data type.

   a) i > 0
   b) i = 0
   c) i++
   d) (float)i
   e) i == 0
   f) a String" + i

2) Consider the following expression:
   i--%5>0

   What is the result of the expression, assuming that the value of i is initially 10?

3) Modify the expression of Ex 2 so that it has the same result but is easier for programmers to read.

4.4 CONTROL STATEMENTS

The program is a set of statements, which are stored into a file. The interpreter executes these statements in a sequential manner, i.e., in the order in which they appear in the file. But there are situations where programmers want to alter this normal sequential flow of control. For example, if one wants to repeatedly execute a block of statements or one wants to conditionally execute statements. Therefore' one more category of statements called control flow statements is provided. In *Table 3*, different categories of control statements are defined:

Note: Although goto is a reserved word, currently the Java programming language does not support the goto statement.

Table 3: Control flow statements

Statement Type	Keyword
Selection	If-else, switch-case
Iteration	While, do-while, for
Jump	Break, continue, label;, return
Exception handling	Try-catch-finally, throw

We will cover the *first three* types of statements in the next three sections. The last statement type will be covered in Unit 4 Block 2 of this course.

4.5 SELECTION STATEMENTS

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

The if / else Statements

The " **if** " statement enables the programmer to selectively execute other statements in the program, based on some condition. The code block governed by the " if " is executed if a condition is true. Generally, the simple form of it can be written like this:

```
if (expression)
{
   statement(s)
}
```

The following code block illustrates the ***if*** statement:

```
Public class If Example
 {
   public static void main (String[] args)
   {
      int x,y;
      x = 10; y = 50;
      if (x < y)
         System.out.println("x is less than y" );
      if (x > y)
         System.out.println ("x is greater than y");
      if (x = y)
         System.out.println ("x is equal to y");
    }
   }
```

If a programmer wants to perform a different set of statements when the expression is false, s/he can use the ***else*** statement.

Let us consider another example. The following code performs different actions depending on whether the user clicks the OK button or another button in an alert window. To do this, programmer can use **if** statement along with an **else** statement. The "*else code block"* is executed only-if "*if code block*" is false.

```
// response is either OK or CANCEL depending
// on the button pressed by the user
if (response == OK)
{
// code to perform OK action
}
else
{
// code to perform Cancel action
}
```

There is one more form of the else statement "**else if "**. It executes a statement based on another expression. "If statement" can have any number of companion *else if* statements but at the end, only one else part will be there. Let us consider the following example in which the program assigns a grade based on the value of a test score: an **A** for a score of 90% or above, a **B** for a score of 75% or above, and so on:

```
public class If Else Example
{
   public static void main (String[] args)
   {
      int test score = 78;
```

```
        char grade;
        if (test score >= 90)
        {
            grade = 'A';
        }
       else if (testscore >= 75)
        {
            grade = 'B';
        }
       else if (testscore >= 60)
       {
            grade = 'C';
        }
       else if (testscore >= 50)
      {
            grade = 'D';
      }
      else
      {
           grade = 'F';
      }
     System.out.println ("Grade = " + grade);
     }
}
```

The output from this program is:

Grade = B

The value of testscore can satisfy more than one of the expressions in the compound if statement: 78 >= 75 and 78 >= 60. At runtime, the system processes a compound if statement by evaluating the conditions. Once a condition is satisfied, the appropriate statements are executed (i.e. grade = 'B';), and control passes out of the "if statement" without evaluating the remaining conditions.

The?: Operator

The Java programming language also supports an operator"**?:**" known as ternary operator, this operator is a compact version of an *if statement*. It is used for making two-way decisions. It is a combination of ?and :, and takes three operands. It is popularly known as the conditional operator. The general form of use of the *conditional operator* is as follows:

Conditional expression **?** expression**:** expression2

The *conditional expression* is evaluated first. If the result is true, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned.

```
if (x <= 10)
commission = 10 * x;
else
commission = 15 * x;
System.out.println("The Total commission is " + commission );
```

You could rewrite this statement using the?: operator:

```
commission =  (x <= 10)? (10 * x): (15 * x);
```

System.out.println ("The Total commission is " + commission);} + "case.");
When the conditional operator is used, the code becomes more *concise* and *efficient*.

The switch Statement

Java has a built-in multiway decision statement known as **switch** statement. The *switch* statement is used to conditionally perform statements based on an integer expression. Let us consider the following example. Using the *switch* statement, it displays the name of the month, based on the value of *month* variable.

```
public class SwitchExample1
{
   public static void main(String[] args)
   {
      int month = 11;
      switch (month)
        {
          case 1:  System.out.println("January"); break;
          case 2:  System.out.println("February"); break;
          case 3:  System.out.println("March"); break;
          case 4:  System.out.println("April"); break;
          case 5:  System.out.println("May"); break;
          case 6:  System.out.println("June"); break;
          case 7:  System.out.println("July"); break;
          case 8:  System.out.println("August"); break;
          case 9:  System.out.println("September"); break;
          case 10: System.out.println("October"); break;
          case 11: System.out.println("November"); break;
          case 12: System.out.println("December"); break;
        }
     }
}
```

The *switch* statement evaluates its expression and executes the appropriate *case* statement. Thus, in this case it evaluates the value of month and the output of the program is: November. It can be implemented by using an if statement:

```
int month = 11;
if (month == 1)
{
   System.out.println ("January");
}
 else if (month == 2)
{
   System.out.println("February");
}. . . // and so on
```

You can decide which statement (*if or switch*) to use, based on readability and other factors.

- An "**if statement**" can be used to make decisions based on ranges of values or conditions.
- A **switch** statement can make decisions based only on a single integer value. The value provided to each case statement must be unique.

It is important to notice the use of the **break** statement after each case in the switch statement. Each break statement terminates the enclosing **switch** statement, and the flow of control continues with the first statement following the switch block.

The **break** statements are necessary because without them, the case statements fall through. That is, without an explicit break, control will flow sequentially through subsequent case statements.

Let's consider the following example.

```
public class SwitchExample2
 {
   public static void main(String[] args)
   {
       int month = 2;
       int year = 2003;
       int numDays = 0;
       switch (month)
       {
           case 1:
           case 3:
           case 5:
           case 7:
           case 8:
           case 10:
           case 12:
              numDays = 31;
              break;
           case 4:
           case 6:
           case 9:
           case 11:
              numDays = 30;
                 break;
           case 2:
              if ( ((year % 4 == 0) && !(year % 100 == 0))
                  || (year % 400 == 0) )
                 numDays = 29;
              else
                 numDays = 28;
              break;
       }
        System.out.println("Number of Days = " + numDays);
   }
 }
```

The output from the above program is: Number of Days = 28

Now let us see how **break** is used to terminate loops in branching statements and the use of default statement at the end of the switch to handle all values that aren't explicitly handled by one of the case statements.

```
int month = 11;
. . .
switch (month)
{
   case 1:  System.out.println("January"); break;
   case 2:  System.out.println("February"); break;
   case 3:  System.out.println("March"); break;
   case 4:  System.out.println("April"); break;
   case 5:  System.out.println("May"); break;
   case 6:  System.out.println("June"); break;
   case 7:  System.out.println("July"); break;
   case 8:  System.out.println("August"); break;
```

```
    case 9:  System.out.println("September"); break;
    case 10: System.out.println("October"); break;
    case 11: System.out.println("November"); break;
    case 12: System.out.println("December"); break;
    default: System.out.println("It is not a valid month!"); break;
 }
```

Now let us see the iterative statements.

4.6 ITERATIVE STATEMENTS

Java iteration statements are **while**, **do-while** and **for**. These statements enable program execution to repeat one or more statements, i.e. they create **loops**. A loop repeatedly executes the same set of instructions until a termination condition is met.

The while and do-while Statements

A **while** statement is used to continually execute a block of statements provided a condition remains true. The general syntax of the *while* statement is:

```
while (expression)
{
  //body of loop
  statement
}
```

Here first, the while statement evaluates *expression*, which returns a boolean value. If the expression returns *true*, then the while statement executes the statement(s) in the body of the loop. The while statement continuously keeps on testing the *expression* and executing the code block until the expression returns *false*.

The example program shown below, displays the table of 6 unto 10 count. Every time while statement checks the value of count variable and displays the line for the table until the value of count becomes *greater than 10.*

```
public class While Example
{
   public static void main (String[] args)
   {
      int count = 1;
      int product;
     System.out.println (“Table of 6”);
      while (count <= 10)
      {
        product = 6 * count;
        System.out.println (“ 6 x “ + count + “ = “ + product);
        count++;
      }
   }
}
```

The do-while statement

The Java programming language also provides another statement which is similar to the while statement; the *do-while* statement. The general syntax of the do-while is:

```
do
{
   statement(s)
```

} while (*expression*);

The *do-while* evaluates the expression at the bottom instead of evaluating it at the top of the loop. Thus the code block associated with a *do-while* is executed at least once.

Here is the previous example program rewritten using **do-while**:

```
public class Do While Example
    {
   public static void main (String[ ] args)
   {
      int count = 1;
      int product;
      System.out.println("Table of 6");
      do
      {
         product = 6 * count;
         System.out.println(" 6 x " + count + " = " + product);
         count++;
      } while (count <= 10);
   }
}
```

The for Statement

The **for** statement provides a way to iterate over a range of values. The general form of the **for** statement can be expressed as:

```
for (initialization; termination; increment)
 {
   statement
 }
```

- The *initialization* is an expression that initializes the loop. It is executed only once at the beginning of the loop.
- The *termination* expression determines when to terminate the loop. This expression is evaluated at the top of iteration of the loop. When the expression evaluates to *false*, the loop terminates.
- The *increment* is an expression that gets invoked after each iteration through the loop. All these components are optional.

Often **for** loops are used to iterate over the elements in an array, or the characters in a string. The following example uses a **for** statement to iterate over the elements of an array and print them:

```
public class For Example
{
   public static void main (String[ ] args)
   {
      int[ ] array O fInts = { 33, 67, 31, 5, 122, 77,204, 82, 163, 12, 345, 23 };
      for (int i = 0; i < arrayOfInts.length; i++)
      {
       System.out.print (array O fInts [i] + " ");
      }
       System.out.println();
   }
}
```

The output of this program is: 33 67 31 5 122 77 204 82 163 12 345 23.

The programmer can declare a local variable within the *initialization* expression of a **for** loop. The scope of this variable extends from its declaration to the end of the block. It is recommended to declare the variable in the initialization expression in case the variable that controls a **for** loop is not needed outside the loop. Declaring the variables like i, j, and k within the for loop initialization expression limits their life span and reduces errors.

4.7 JUMP STATEMENTS

The Java language supports three jump statements:

- The break statement
- The continue statement
- The return statement.

These statements transfer control to another part of the program.

The break Statement

The **break** statement has two forms: *labeled* and *unlabeled*. A *label* is an identifier placed before a statement. The label is followed by a colon (:) like Statement Name: Java Statement;

Unlabeled Form: The unlabeled form of the break statement is used with *switch*. You can note in the given *example*, an **unlabeled break** terminates the enclosing switch statement, and flow of control transfers to the statement immediately following the switch. The unlabeled form of the break statement is *also used* to terminate a for, while, or do-while loop. Let us consider the following example program where a break is used in for loop:

```
public class BreakExample
  {
  public static void main (String[ ] args)
  {
     int [ ] arrayOfInts = {33, 67, 31, 5, 122, 77, 204, 82, 163, 12, 345, 23};
     int search for = 122;
     int i = 0;
     boolean foundIt = false;
     for (; i < arrayOfInts.length; i++)
     {
        if (arrayOfInts[i] == searchfor)
        {
           foundIt = true;
           break;
        }
     }
     if (foundIt)
    {
     System.out.println( searchfor + " at index " + i);
     }
    else
     {
     System.out.println(searchfor + "not in the array");
     }
  }
}
```

The **break** statement terminates the “for” loop when the value is found. The flow of control transfers to the statement following the enclosing for, which is the print statement at the end of the program.

The output of this program is:

122 at index 4

The unlabeled form of the break statement is used to terminate the innermost switch, for, while, or do-while structures.

Labeled Form: The labeled form terminates an **outer statement**, which is identified by the label specified in the break statement. Let us consider the following *example* program, which searches for a value in a two-dimensional array. When the value is found, a *labeled break* terminates the statement labeled search, which is the outer for loop:

```
public class BreakWithLabelExample
 {
   public static void main (String[ ] args)
   {
      int[][] arrayOfInts = { { 33, 67, 31, 5 },{ 122, 77, 204, 82 },{ 163, 12, 345, 23 }
};
       int searchfor = 122;
       int i = 0;
       int j = 0;
       boolean foundIt = false;
      search:
       for ( ; i < arrayOfInts.length; i++)
       {
          for (j = 0; j < arrayOfInts[i].length; j++)
            {
             if (arrayOfInts[i][j] == searchfor)
             {
                foundIt = true;
                break search;
             }
            }
       }
      if (foundIt)
       {
        System.out.println (search for + " at " + i + ", " + j);
       }
     else
       {
       System.out.println (search for + "not in the array");
       }
   }
}
```

The output of this program is: 122 at 1, 0

The break statement terminates the labeled statement; the flow of control transfers to the statement immediately following the labeled (terminated) statement.

The continue Statement

The continue statement is used to skip the current iteration of a for, while, or do-while loop.

The **unlabeled form** of continues statement skips the remainder of this iteration of the loop. It evaluates the boolean expression that controls the loop at the end. Let us take the following *example* program in which a string buffer is checking each letter.

If the current character is not '**s'**, the continue statement skips the rest of the loop and proceeds to the next character. If it is '**s'**, the program increments a counter.

```
public class Continue Example
  {
   public static void main (String[ ] args)
   {
          String Buffer search Str = new String Buffer ("she sell sea-shell on the sea
          shore");
      int max = search Str. length  ();
      int numOfS = 0;
      for (int i = 0; i < max; i++)
      {
         if (search Str. charAt(i) != 's') // we want to count only S's
            continue;
         numOfS++;
         }
      System.out.println("Found " + numOfS + " S's in the string:");
      System.out.println (search Str);
   }
}
```

Here is the output of this program:

```
Found 6 S's in the string.
she sell sea-shell on the sea shore
```

The **labeled form** of the continue statement skips the current iteration of an outer loop marked with the given label. The following *example* program uses *nested loops* to search a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. This program uses the labeled form of continue to skip an iteration in the outer loop:

```
public class ContinueWithLabelExample
{
 public static void main (String[ ] args)
  {
  String searchMe = "Look for a substring in me";
  String substring = "sub";
  boolean foundIt = false;
  int max = searchMe.length() - substring.length();
 test:
  for (int i = 0; i <= max; i++)
  {
     int n = substring.length();
     int j = i;
     int k = 0;
     while (n-- != 0)
     {
        if (searchMe.charAt(j++) != substring.charAt (k++))
        {
          continue test;
        }
     }
          found It = true;
          break test;
  }
  System.out.println (found It? "Found it": "Didn't find it");
}
}
```

Here is the output from this program: Found it

The return Statement

The last of Java's jump statements is the **return** statement. It is used to *explicitly* return from the current method. The flow of control transfers *back to the caller* of the method. The return statement has **two forms**: one that returns a value and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the *return* keyword:

return ++count;

The data type of the value returned by **return** must match the type of the method's declared return value. When a method is declared *void*, use the form of return that doesn't return a value like "return;"

☞ Check Your Progress 2

1) **State True or False for the followings**

a) The modulus operator (%) can be only used with integer operand. ☐

b) If a = 10 and b = 15, then the statement x = (a > b)? a: b; assigns the value 15 to x. ☐

c) In evaluating a logical expression of type BoolExp1 && BoolExp2 Both the Boolean expressions are not always evaluated. ☐

d) In evaluating the expression (x == y && a < b) the boolean expression x == y is evaluated first and then a < b is evaluated. ☐

e) The default case is always required in the switch selection structure. ☐

f) The break statement is required in the default case of a switch selection structure. ☐

g) A variable declared inside for the loop control cannot be referenced outside the loop. ☐

h) The following loop construct is valid ☐

```
int i = 10;
while (i)
{
Statements
}
```

i) The following loop construct is valid ☐

```
int i = 1; sum = 0;
do {Statements }
while ( i < 5 || sum < 20);
```

2) **Select appropriate answer for the followings:**

a) What will be the value of x , i and j after the execution of following statements?

```
int x , i , j;
i = 9;
j = 16;
x = ++i + j++
```

(i) x = 25, i = 9, j = 16
(ii) x = 26, i = 10, j = 17
(iii) x = 27, i = 9, j = 16
(iv) x = 27, i = 10, j = 17

b) If i and j are integer type variables, what will be the result of the expression

```
i % j
```

when i = 7 and j = 2?

(i) 0
(ii) 1
(iii) 2
(iv) None of the above

c) If i and j are integer type variables, what will be the result of the expression
i % j
when i = – 16 and j = –3 ?

(i) -1
(ii) 1
(iii) 5
(iv) None of the above

d) Consider the following code:

```
Char c = 'a';
Switch (c)
{
case 'a' :
System.out.println( "A");
case 'b' :
System.out.println( "B");
default :
System.out.println( "C");
}
```

3) Which of the following statements is True?

(i) Output will be A ☐
(ii) Output will be A followed by B ☐
(iii) Output will be A, followed by B, then followed by C ☐
(iv) Illegal code. ☐

4) What is wrong with the following program code:

```
Switch (x)
{
case 1:
m = 15;
n = 20;
case 2:
p = 25;
break;
y = m + n – p;
}
```

5) How is the if … else if combination more general than a switch statement?

4.8 ARRAYS

An *array* is an important data structure supported by all the programming languages. An array is a *fixed-length* structure that stores multiple values of the same type. You can group values of the same type within arrays. *For example* array name salary can be used to represent a set of salaries of a group of employees. Arrays are supported directly by the Java programming language but there is no array class.
The length of an array is established when the array is created (at runtime). After creation, an array is a fixed-length structure.

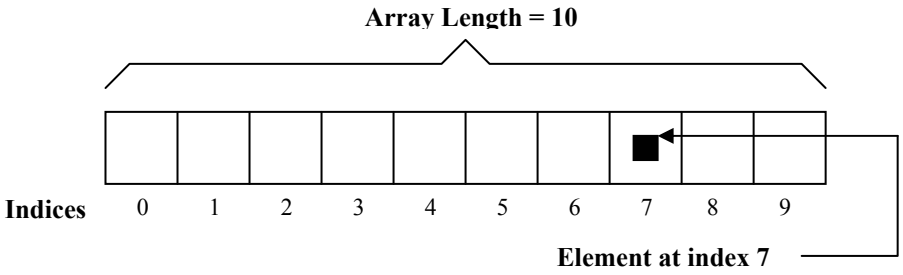

Figure 1: Example of an array

An *array element* is one of the values within an array and is accessed by its position within the array. Index counting in Java starts from 0. For example, to access the salary of 8th employee in array salary, it is represented by *salary[7]* as given in *Figure 1.*

Here is an example program that creates the array, puts some values in it, and displays the values.

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

Declaring a Variable to refer to an Array

The following line of code from the sample program declares an array variable:

Int [] an Array; // declare an array of integers

An array declaration has two components:

- Array's type
- Array's name.

An **array's type** is written *type* [], where *type* is the data type of the elements contained within the array, and [] indicates that this is an array. The example program uses **int []** and name of the array is an Array. Here, an Array will be used to hold integer data. Here are some other *declarations* for arrays that hold other types of data:

```
float[] marks;
boolean [] gender;
Object[] listOfObjects;
String[] NameOfStudents;
```

The *declaration* for an array variable does not allocate any memory to contain the array elements. The example program must assign a value to an Array before the name refers to an array.

Creating an Array

The programmer creates an array explicitly using Java's new operator. The next statement in the example program allocates an array with enough memory for ten integer elements and assigns the array to the variable an Array declared earlier.

An Array = new int[10]; // create an array of integers

In general, when creating an array, the programmer uses the new operator, followed by data type of the array elements, and then followed by the number of elements desired enclosed within square brackets ('[' and ']') like:

new element Type [array Size]

If the new statement were omitted from the example program, the compiler would print an error and compilation would fail.

Accessing an Array Element

Let us consider the following code, which assigns values to the array elements:

```
for (int i = 0; i < anArray.length; i++)
 {
anArray[i] = i;
System.out.print (anArray[i] + " ");
 }
```

It shows that to reference an array element, append square brackets to the array name. The value between the square brackets indicates (either with a variable or some other expression) the *index* of the element to access. As mentioned earlier in Java, array indices begin at 0 and end at the array length minus 1.

Getting the Size of an Array

You can get the size of an array, by writing "***arrayname*.length".** Here, length is not a method, it is a property provided by the Java platform for all arrays. The "for" loop in our example program iterates over each element of an Array, assigning values to its elements. The "for" loop uses the **anArray.length** to determine, when to terminate the "for" loop.

Array Initializers

The Java programming language provides one another way also for creating and initializing an array. Here is an example of this syntax:

boolean [] answers = {true, false, true, true, false};

The number of values provided between {and} determines the length of the array.

Multidimensional array in Java

In Java also you can take multidimensional arrays as arrays of arrays. You can declare a multidimensional array variable by specifying each additional index with a set of square brackets. For example

float two Dim [][]= new float [2][3];

This declaration will allocate a 2 by 3 array of float to two Dim. Java also provides the facility to specify the size of the remaining dimensions separately except the first dimension.

To understand the above stated concept see the program given below in which different second dimension size is allocated manually.

```
class Arra_VSize
{
public static void main (String args[])
{
        int i, j, k=0;
        int  twoDim [][] = new int [3][];
        twoDim[0] = new int[1];
        twoDim[1] = new int[2];
        twoDim[2] = new int[3];
        for ( i= 0; i <3 ; i++)
        {
          for (j = 0; j< i+1;  j++)
                {
                twoDim[i][j] = k +   k*3;
                k++;
                }
        }
  for ( i= 0; i <3 ; i++)
        {
          for (j = 0; j< i+1;  j++)
                System.out.print(twoDim[i][j] + " ");
                System.out.println();
        }
}
}
```

Output of this program

```
0
4 8
12 16 20
```

☞ Check Your Progress 3

1) Why should switch statement be avoided?

2) Answer the following questions using the sample code given below:

```
String[] touristResorts = {     "Whistler Blackcomb", "Squaw Valley",
"Ooty",  "Snowmass", "Flower Valley", "Taos"};
```

   a) What is the index of "Ooty" in the array?
   b) Write an expression that refers to the string Ooty within the array.
   c) What is the value of the expression touristResorts.length?
   d) What is the index of the last item in the array?
   e) What is the value of the expression touristResorts[4]?

3) The following program contains a bug. Find it and fix it.

```
//
// This program compiles but won't run successfully.
public class WhatHappens {
public static void main(String[] args) {
StringBuffer[] stringBuffers = new StringBuffer[10];
for (int i = 0; i < stringBuffers.length; i ++) {
stringBuffers[i].append("StringBuffer at index " + i);
}
}
}
```

4.9 SUMMARY

In this unit we first discussed about **expression**, which is a series of variables, operators, and method calls that evaluates to a single value. You can write compound expressions by combining expressions as long as the types required by all of the operators involved in the compound expression are correct. But remember Java platform evaluates the compound expression in the order dictated by *operator precedence* .We discussed three kinds of statements: expression statements, declaration statements, and control flow statements. The second important concept we discussed is that for controlling the flow of a program, Java has **three loop** constructs, Let us summarize these constructs:

Loop constructs

- Use the **while** statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the *top of the loop*.
- Use the **do-while** statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the *bottom of the loop*, so the statements within the do-while block execute at least once.
- The **for** statement loops over a block of statements and includes an initialization expression, a termination condition expression, and an increment expression.

Decision-Making Statements

- In the case of the basic **if** statement, a single statement block is executed if the boolean expression is true.
- In case of an **if** statement with a companion **else** statement, the if statement executes the first block if the boolean expression is true; otherwise, it executes the second block of code.
- Use **else if** to construct compound if statements.
- Use **switch** to make multiple-choice decisions based on a single integer value. It evaluates an integer expression and executes the appropriate case statement.

Jump Statements

Jump statements change the *flow of control* in a program to a labeled statement. You label a statement by placing a legal identifier (the label) followed by a colon (:) before the statement.

- Use the **unlabeled** form of the **break** statement to terminate the innermost switch, for, while, or do-while statement.
- Use the **labeled** form of the **break** statement to terminate an outer switch, for, while, or do-while statement with the given label.
- A **continue** statement terminates the current iteration of the innermost loop and evaluates the boolean expression that controls the loop.
- The **labeled** form of the **continue** statement terminates the current iteration of the loop with the given label.
- Use **return** to terminate the current method.

And in the end we discussed an **array,** which is a *fixed-length data structure* that can contain multiple objects of the same type. An element within an array can be accessed by its index. Indices begin at 0 and end at the length of the array.

4.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) a) i > 0 [boolean]
b) i = 0 [int]
c) i++ [int]
d) (float)i [float]
e) i == 0 [boolean]
f) "aString" + i [String]

2) false

3) (i-- % 5) > 0

Check Your Progress 2

1) State True or false

a) False
b) True
c) True
d) True
e) False
f) False
g) False
h) False
i) True

2) (a) (ii)
(b) (ii)
(c) (i)
(d) (ii)

3) Statement y = m + n – p; is unreachable.

4) The switch statement must be controlled by a single integer control variable and each case section must correspond to a single constant value for the variable. The if … else if combination allows any kind of condition after each if. Consider the following example:

```
…
if (income < 50000)
   System.out.println ( “Lower income group, Tax is nil”);
else if (income < 60000)
   System.out.println ( “Lower Middle income group, Tax is below 1000”);
else if (income < 120000)
   System.out.println ( “Middle income group, Tax is below 19000 ”);
else
   System.out.println ( “Higher income group”);
```

5) It refers to the way the switch statements executes its various case sections. Each statement that follows the selected case section will be executed unless a break statement is encountered.

Check Your Progress 3

1) Sometimes it is easy to fall through accidentally to an unwanted case while using switch statement. It is advisable to use `if`/`else` instead.

2)

a) 2.
b) Tourist Resorts[2].
c) 6.
d) 5.
e) Flower Valley

3)

The program generates a Null Pointer Exception on line 6. The program creates the array, but does not create the string buffers, so it cannot append any text to them. The solution is to create the 10 string buffers in the loop with new String Buffer() as follows:

```
public class ThisHappens
{
public static void main(String[] args)
{
StringBuffer[] stringBuffers = new StringBuffer[10];

for (int i = 0; i < stringBuffers.length; i ++)
{
stringBuffers[i] = new StringBuffer();
tringBuffers[i].append("StringBuffer at index " + i);
}
}
}
```