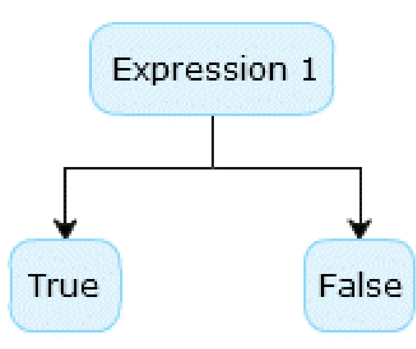
Session 4
Programming Constructs and Arrays

Objectives

- Explain selection constructs
- Explain selection constructs
- Describe loop constructs
- Explain jump statements in C#
- Define and describe arrays
- List and explain the types of arrays
- Explain the Array class

Selection Constructs

- A selection construct:
  - Executes a particular block of statements based on a boolean condition, which is an expression returning true or false.
  - Allow you to take logical decisions about executing different blocks of a program to achieve the required logical output.
- Decision-making constructs:
  - if…else
  - if…else…if
  - Nested if
  - switch…case



The `if` Statement 1-2

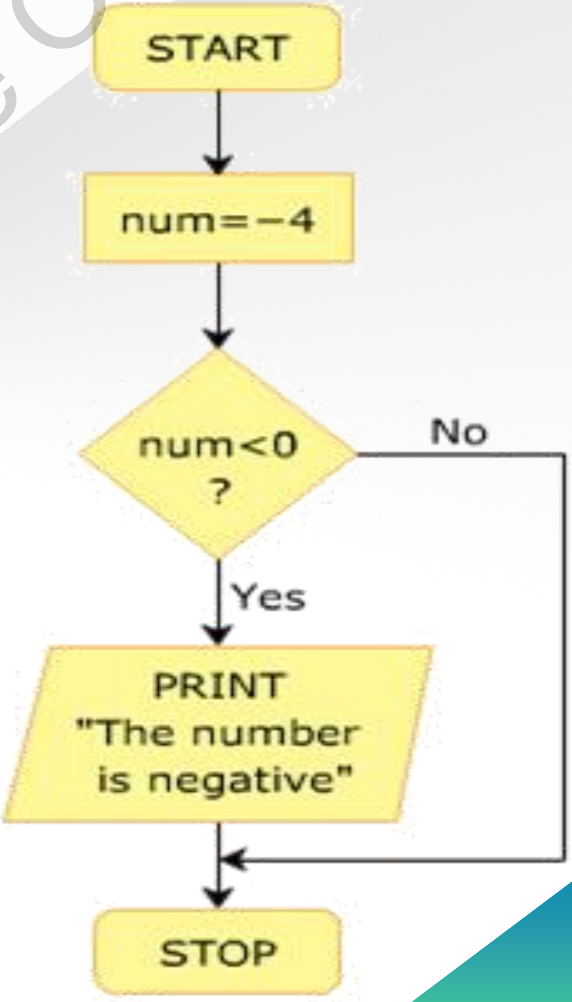
- The **if** statement allows you to execute a block of statements after evaluating the specified logical condition.

Syntax

```
if (condition)
{
   // one or more statements;
}
```

where,

- condition: Is the boolean expression.
- statements: Are set of executable instructions executed when the boolean expression returns true



The `if` Statement 2-2

Snippet

```
int num = -4;
if (num < 0)
{
   Console.WriteLine(“The number is negative”);
}
```

Output

```
The number is negative.
```

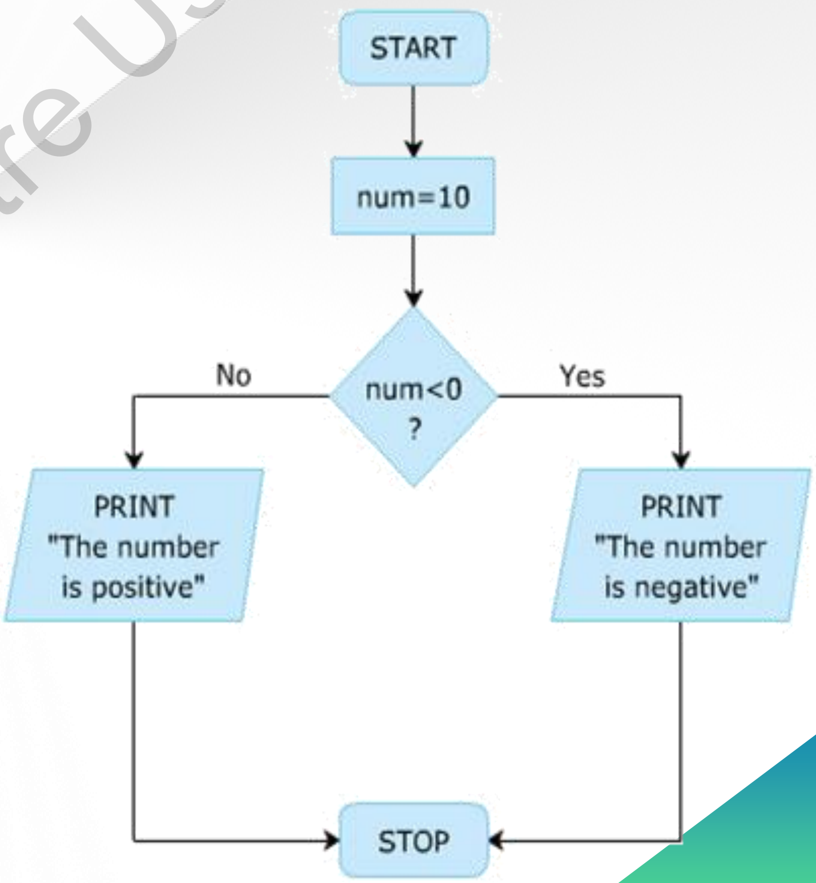
The `if…else` Construct 1-2

- The **if…else** construct starts with the if block followed by an else block and the else block starts with the else keyword followed by a block of statements.

Syntax

```
if (condition)
{
// one or more statements;
}
else
{
//one or more statements;

}
```



The if…else Construct 2-2

Snippet

```
int num = 10;
if (num < 0)
{
   Console.WriteLine(“The number is negative”);
}
else
{
   Console.WriteLine(“The number is positive”);
}
```

Output

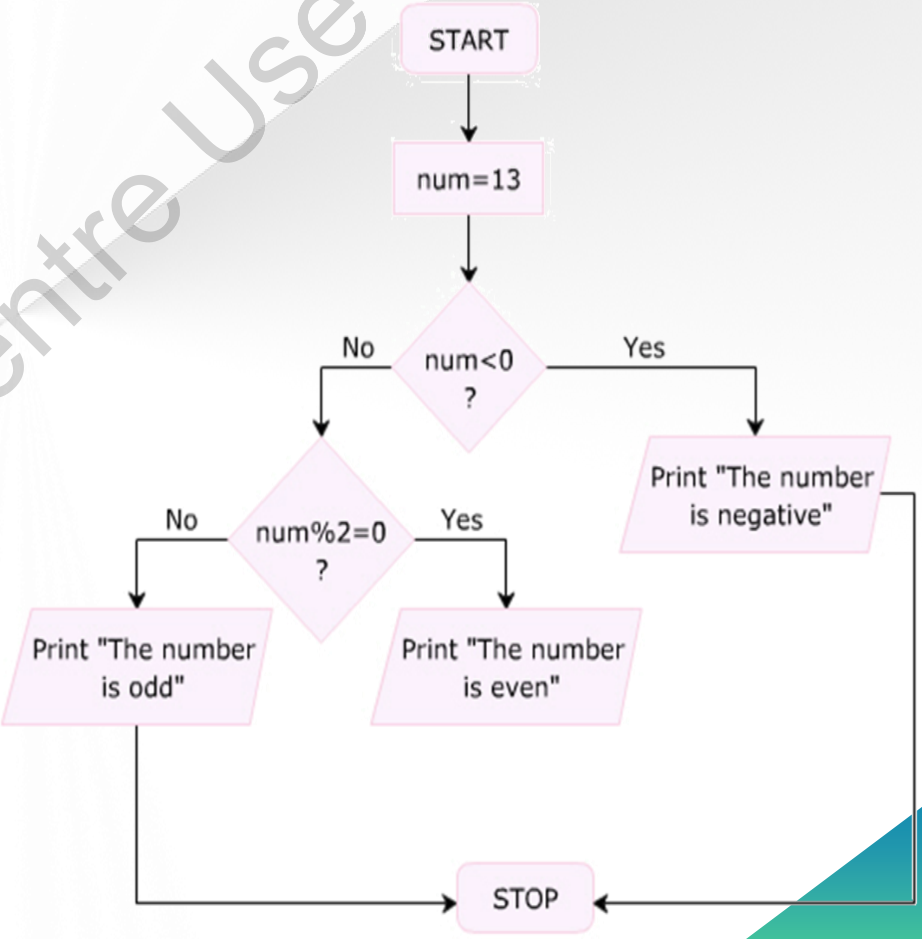
```
The number is positive.
```

The `if…else…if` Construct 1-2

- The **if…else…if** construct allows you to check multiple conditions to execute a different block of code for each condition.

Syntax

```
{
// one or more statements;
}
else if (condition)
{
// one or more statements;
}
else
{
// one or more statements;
}
```



The if…else…if Construct 2-2

Snippet

```
int num = 13;
if (num < 0)
{
  Console.WriteLine(“The number is
negative”);
}
else if ((num % 2) == 0)
{
  Console.WriteLine(“The number is even”);
}

else
{
  Console.WriteLine(“The number is odd”);
}
```

Output

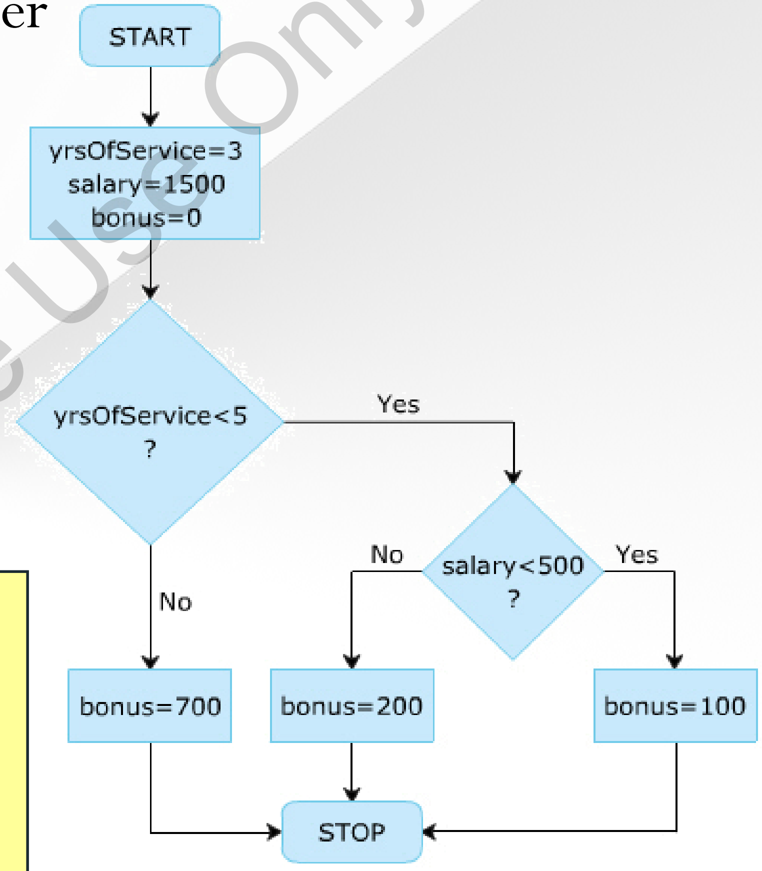
```
The number is odd.
```

Nested `if` Construct 1-2

- Consists of multiple `if` statements, the outer if statement and the inner `if` statements.
- The outer `if` condition controls the execution of the inner `if` statements

Syntax

```
if (condition)
{
    // one or more statements;
    if (condition)
    {
        // one or more statements;
        if (condition)
        {
        // one or more statements;
        }
    }
}
```



Nested `if` Construct 2-2

Snippet

```
int yrsOfService = 3;
double salary = 1500;
int bonus = 0;
if (yrsOfService < 5)
{
    if (salary < 500)
    {
       bonus = 100;
    }
    else
    {
       bonus = 200;
    }
    }
    else
    {
       bonus = 700;
    }

Console.WriteLine(“Bonus amount: ‟ + bonus);

}
```

Output

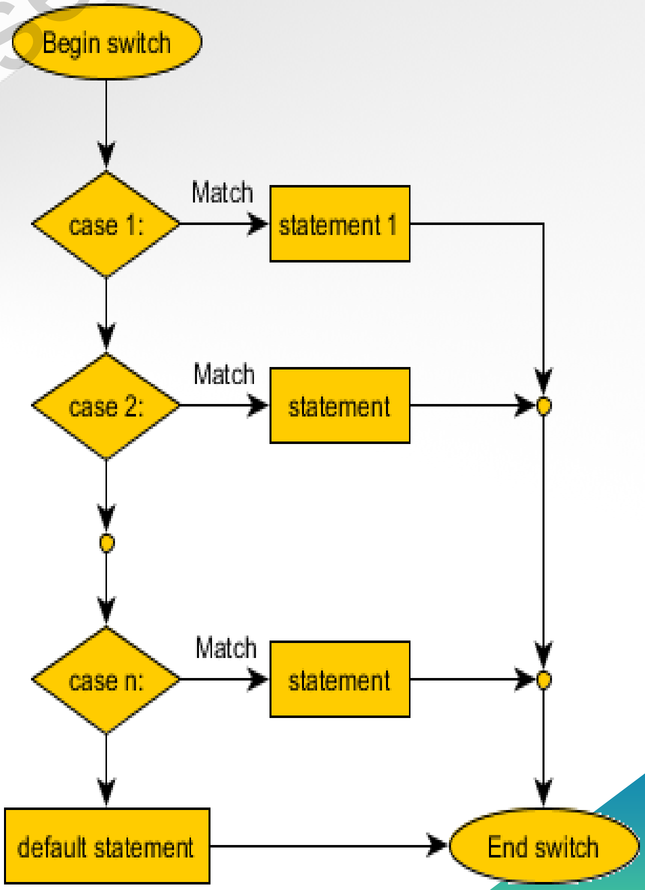
```
Bonus amount: 200
```

switch…case Construct 1-2

- A program is difficult to comprehend when there are too many if statements representing multiple selection constructs.

Syntax

```
switch (n)
{
    case 1:
    // code to be executed if n = 1;
    break;//to terminate a statement
    case 2:
    // code to be executed if n = 2;
    break;
    default:
    // code to be executed if n
    // doesn't match any cases
}
```



switch…case Construct 2-2

Snippet

```
int day = 5;
switch (day)
{
    case 1:
       Console.WriteLine(“Sunday”);
       break;
    case 2:
       Console.WriteLine(“Monday”);
       break;
    case 3:
       Console.WriteLine(“Tuesday”);
       break;

    case 4:
       Console.WriteLine(“Wednesday”);
       break;
    case 5:
       Console.WriteLine(“Thursday”);
       break;
    case 6:
       Console.WriteLine(“Friday”);
       break;
    case 7:
       Console.WriteLine(“Saturday”);
       break;
    default:
       Console.WriteLine(“Enter a number between 1 to 7”);
       break;
}
```

Output

```
Thursday
```

Nested `switch…case` Construct 1-2

- A case block of a **switch…case** construct can contain another switch…case construct.

Snippet

```
namespace Samsung
using System;
class Math {
  static void Main(string[] args)    {
        int numOne; int numTwo;
              int result = 0;
              Console.WriteLine("(1) Addition");
              Console.WriteLine("(2) Subtraction");
              Console.WriteLine("(3) Multiplication");
              Console.WriteLine("(4) Division");
              int input = Convert.ToInt32(Console.ReadLine());
              Console.WriteLine("Enter value one");
              numOne = Convert.ToInt32(Console.ReadLine());
              switch (input) {
              case 1:
              result = numOne + numTwo;
              break;
              case 2:
              result = numOne - numTwo;
              break;
              case 3:
              result = numOne * numTwo;
              break;
              case 4:
              Console.WriteLine("Do you want to calculate the quotient or remainder?");
              Console.WriteLine("(1) Quotient");
              Console.WriteLine("(2) Remainder");
```

Nested `switch…case` Construct 2-2

```
int choice = Convert.ToInt32(Console.ReadLine());
switch (choice) {
case 1:
result = numOne / numTwo;
break;
case 2:
result = numOne % numTwo;
break;
default:
Console.WriteLine("Incorrect Choice");
break;
}
break;
default:
Console.WriteLine("Incorrect Choice");
break;
}
Console.WriteLine("Result: " + result);
}
}
```

No-Fall-Through Rule 1-3

In C#, the flow of execution from one case statement is not allowed to continue to the next case statement and is referred to as the 'no-fall-through' rule of C#.

The list of statements inside a case block generally ends with a break or a goto statement, which causes the control of the program to exit the switch…case construct and go to the statement following the construct.

The last case block (or the default block) also has to have a statement like break or goto to explicitly take the control outside the switch…case construct.

C# introduced the no fall-through rule to allow the compiler to rearrange the order of the case blocks for performance optimization.

- Although C# does not permit the statement sequence of one case block to fall through to the next, it does allow empty case blocks (case blocks without any statements) to fall through.

No-Fall-Through Rule 2-3

- A multiple case statement can be made to execute the same code sequence, as shown in following code:

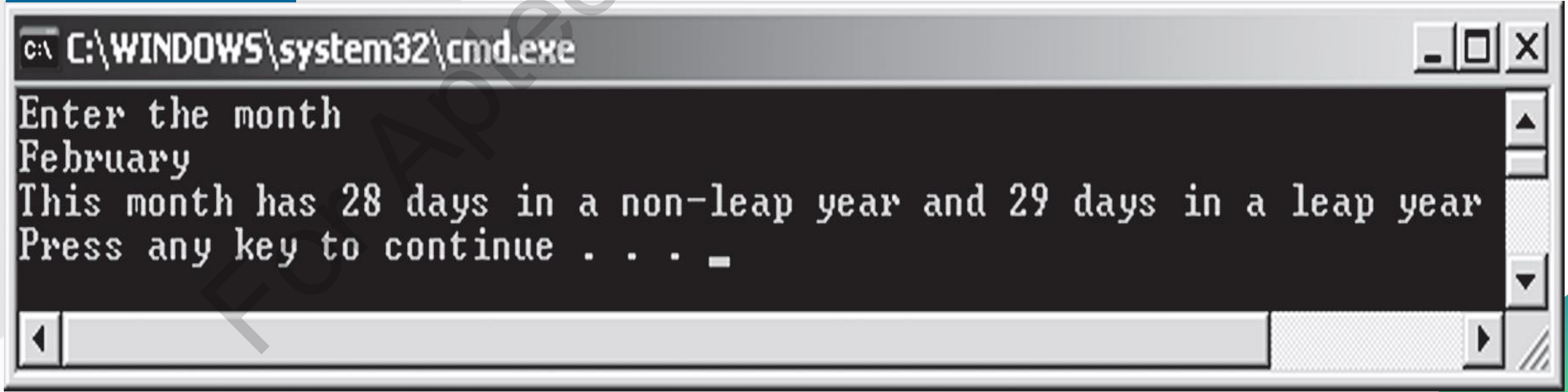
Snippet

```
namespace Samsung
using System;
class Months
{
   static void Main(string[] args)
   {
      string input;
      Console.WriteLine("Enter the month");
      input = Console.ReadLine().ToUpper();
      switch (input)
      {
          case "JANUARY":
          case "MARCH":
          case "MAY":
          case "JULY":
          case "AUGUST":
          case "OCTOBER":
          case "DECEMBER":
          Console.WriteLine ("This month has 31 days");
          break;
          case "APRIL":
          case "JUNE":
          case "SEPTEMBER":
```

No-Fall-Through Rule 3-3

```
            case "NOVEMBER":
            Console.WriteLine ("This month has 30 days");
            break;
            case "FEBRUARY":
            Console.WriteLine("This month has 28 days in
            a non-leap year and 29 days in a leap year");
            break;
            default:
            Console.WriteLine ("Incorrect choice");
            break;
        }
    }
}
```

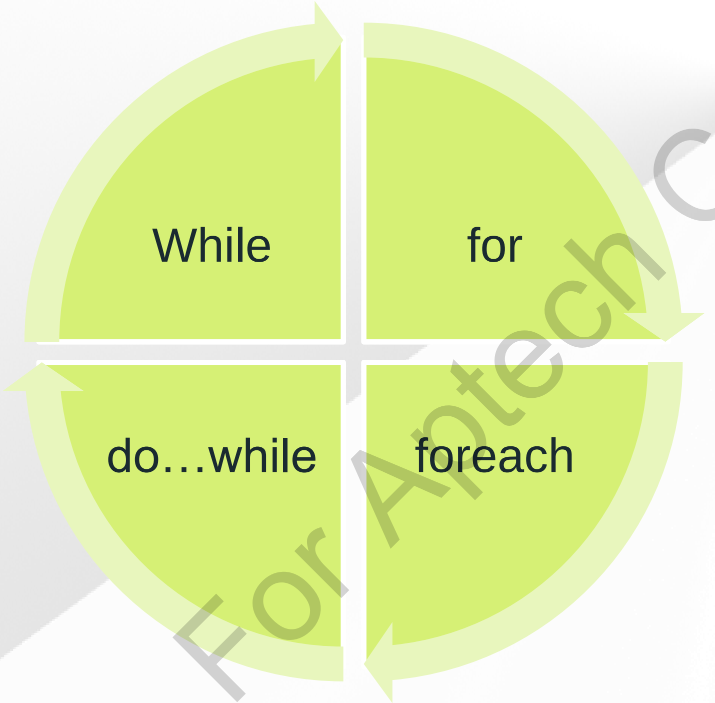
Output

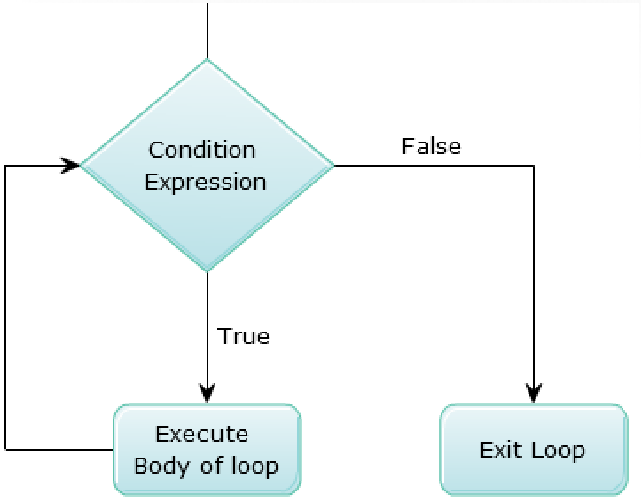


```
Enter the month
February
This month has 28 days in a non-leap year and 29 days in a leap year
Press any key to continue . . .
```

Loop Constructs

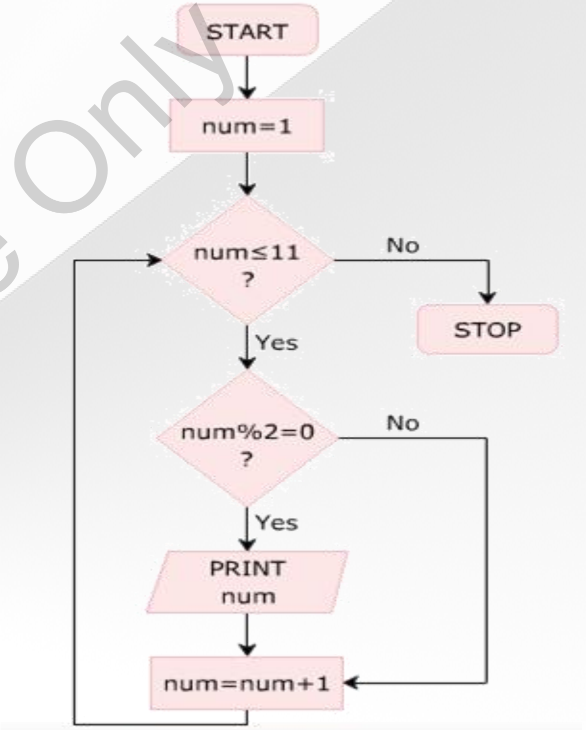
- In software programming, a loop construct contains a condition that helps the compiler identify the number of times a specific block will be executed.
- If the condition is not specified, the loop continues infinitely and is termed as an infinite loop.
- The loop constructs are also referred to as iteration statements.





The `while` Loop

- The while loop is used to execute a block of code repetitively until the condition of the loop remains true.
- condition: Specifies the boolean expression.



Syntax

```
while (condition)
{
  // one or more statements;
}
```

Snippet

```
public int num = 1;
Console.WriteLine(“Even Numbers”);
while (num <= 11)
{
   if ((num % 2) == 0)
   {
       Console.WriteLine(num);
   }
   num = num + 1;
}
```

Output

```
Even Numbers
2
4
6
8
10
```

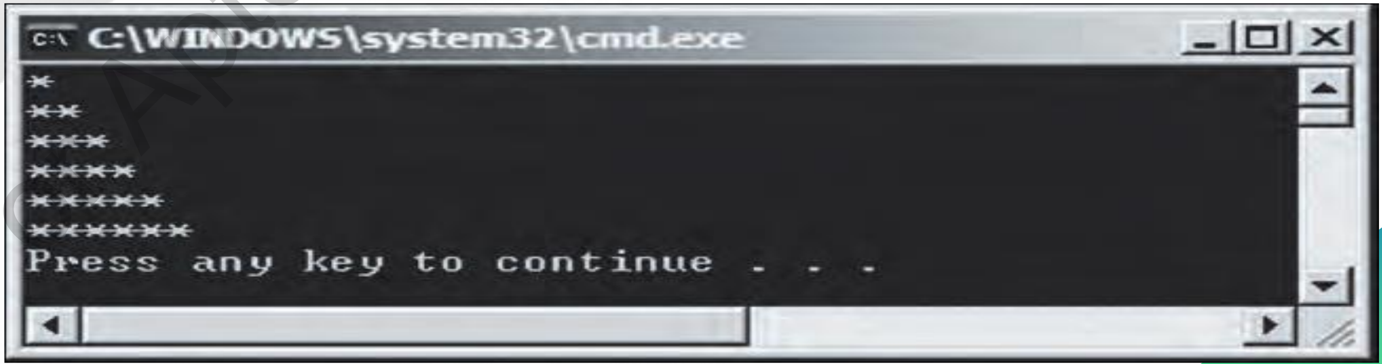
Nested `while` Loop

- A loop within loop

Snippet

```
using System;
class Pattern
{
static void Main(string[] args)
{
   int i = 0;
   int j;
   while (i <= 5)
   {
     j = 0;
     while (j <= i)
     {
     Console.Write("*");
     j++;
     }
     Console.WriteLine();
     i++;
     }
   }
}
```

Output



```
*
**
***
****
*****
******
Press any key to continue . . .
```

The `do-while` Loop

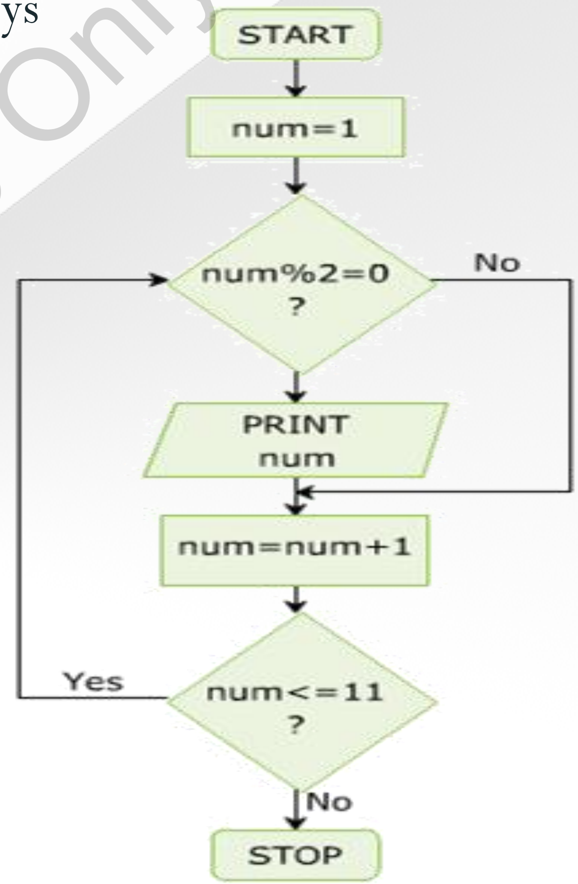
- The do-while loop is like the while loop; however, it is always executed at least once without the condition being checked.

Syntax

```
do
{
// one or more statements;
} while (condition);
```

Snippet

```
int num  =  1;
Console.WriteLine(“EvenNumbers”);
 do
   {
       if  ((num  %  2)  ==  0)
        {
         Console.WriteLine(num);
        }
       num  = num  +  1;
    } while  (num  <=  11);
```

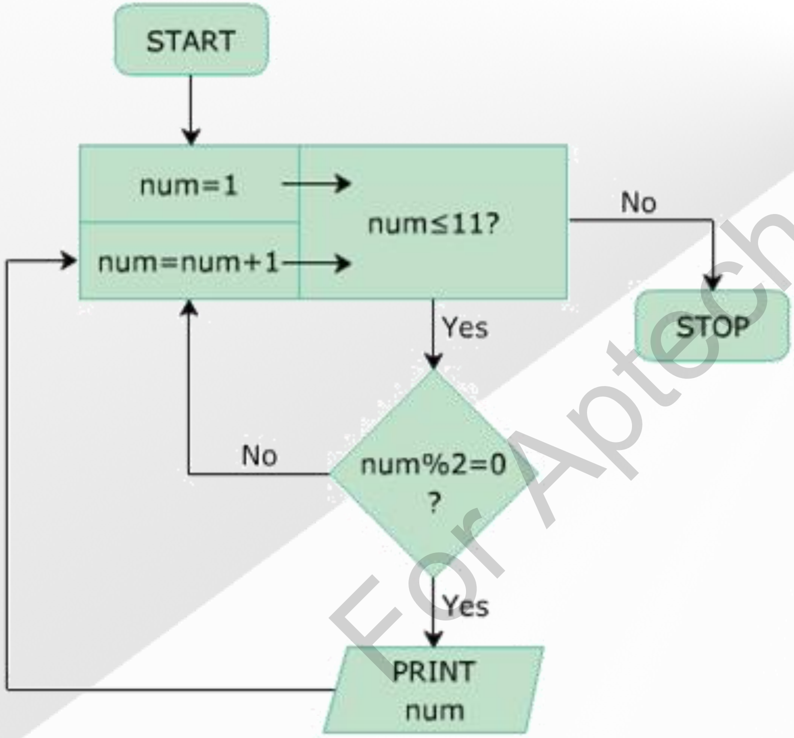


The `for` Loop

- The for statement is like the while statement in its function.
- The statements within the body of the loop are executed until the condition is true.

Syntax

```
for (initialization; condition; increment/decrement)
{
    // one or more statements;
}
```



Snippet

```
int num;
Console.WriteLine(“Even Numbers”);
for (num = 1; num <= 11; num++) {
if ((num % 2) == 0)
{
Console.WriteLine(num);
}
}
```

Nested `for` Loops

- The nested for loop consists of multiple for statements. When one for loop is enclosed inside another for loop, the loops are said to be nested.
- The for loop that encloses the other for loop is referred to as the outer for loop whereas the enclosed for loop is referred to as the inner for loop.
- The outer for loop determines the number of times the inner for loop will be invoked.

Snippet

```
int rows = 2;
int columns = 2;
for (int i = 0; i < rows; i++)
{
   for (int j = 0; j < columns; j++)
   {
      Console.Write(“{0} “,i*j);
   }
   Console.WriteLine();

}
```

The for Loop with Multiple Loop Control Variables

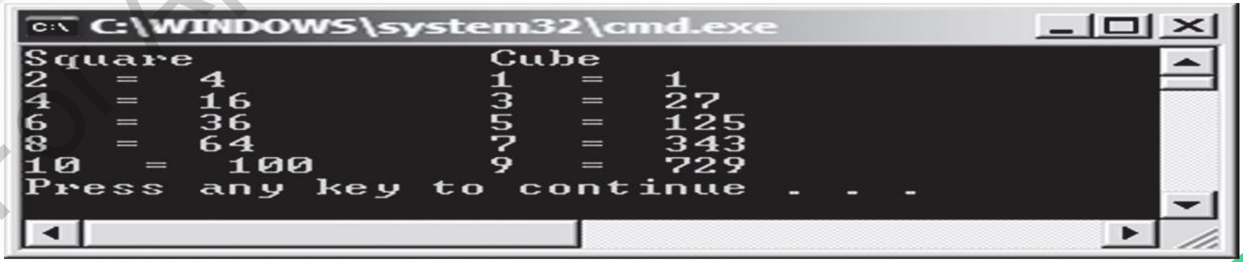
- The for loop allows the use of multiple variables to control the loop.

Snippet

```
using System;
class Numbers
{
    static void Main(string[] args)
    {
        Console.WriteLine("Square \t\tCube");
        for (int i = 1, j = 0; i < 11; i++, j++)
        {
            if ((i % 2) == 0)
            {
                Console.Write("{0} = {1} \t", i, (i * i));
                Console.Write("{0} = {1} \n", j, (j * j * j));
            }
        }
     }

}
```

Output



```
Square                  Cube
2   =   4               1   =   1
4   =   16              3   =   27
6   =   36              5   =   125
8   =   64              7   =   343
10   =   100            9   =   729
Press any key to continue . . .
```

The `for` Loop with Missing Portions

- C# allows the creation of the for loop with all the three portions, the initialization, the conditional expression, and the increment/decrement portion omitted.

Snippet

```
using System;
class Investment
{
      static void Main(string[] args)
      {
         int investment;
         int returns;
         int expenses;
         int profit;
         int counter = 0;
         for (investment=1000, returns=0; returns<investment;)
          {
             Console.WriteLine("Enter the monthly expenditure");
             expenses = Convert.ToInt32(Console.ReadLine());
             Console.WriteLine("Enter the monthly profit");
             profit = Convert.ToInt32(Console.ReadLine());
             investment += expenses;
             returns += profit;
             counter++;
          }
        Console.WriteLine("Number of months to break even: "
       + counter);
       }
}
```

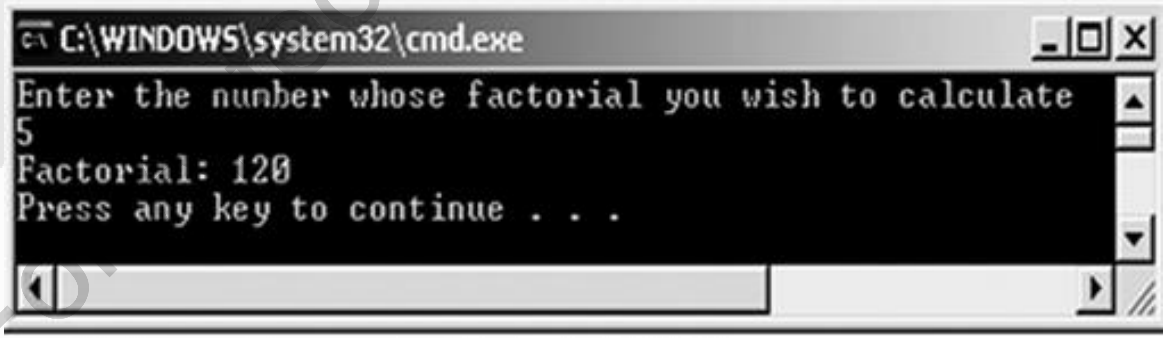
The `for` Loop without a Body

- In C#, a for loop can be created without a body. Such a loop is created when the operations performed within the body of the loop can be accommodated within the loop definition itself.

Snippet

```
using System;
class Factorial
{
    static void Main(string[] args)
    {
      int fact = 1;
      int num, i;
      Console.WriteLine("Enter the number whose factorial you
      wish to calculate");
      num = Convert.ToInt32(Console.ReadLine());
      for (i = 1; i <= num; fact *= i++);
      Console.WriteLine("Factorial: " + fact);
    }
}
```

Output



```
Enter the number whose factorial you wish to calculate
5
Factorial: 120
Press any key to continue . . .
```

Declaring Loop Control Variables in the Loop Definition 1-2

- The loop control variables are often created for loops such as the for loop. Once the loop is terminated, there is no further use of these variables. In such cases, these variables can be created within the initialization portion of the for loop definition.

Syntax

```
foreach (<datatype><identifier> in <list>)
{
    // one or more statements;
}
```

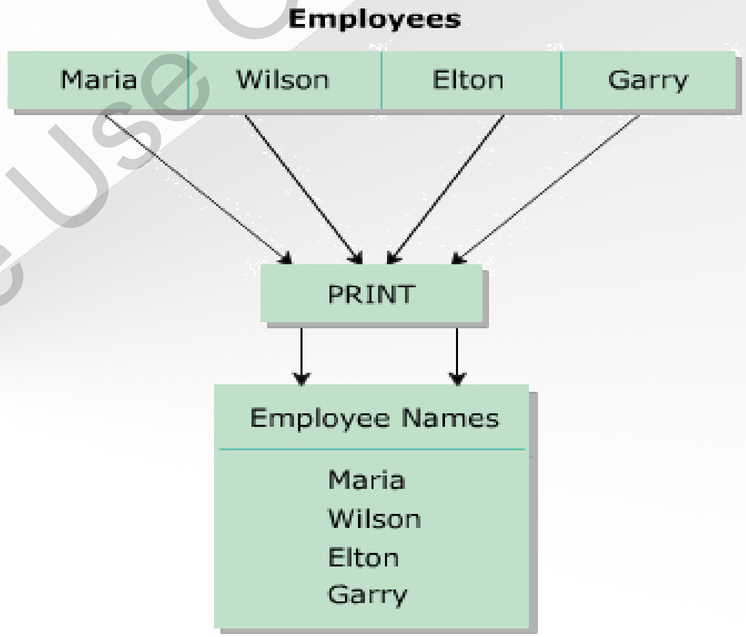
where,

- **datatype:** Specifies the data type of the elements in the list.
- **identifier:** Is an appropriate name for the collection of elements.
- **list:** Specifies the name of the list.

Declaring Loop Control Variables in the Loop Definition 2-2

Snippet

```
string[] employeeNames = { “Maria”,
“Wilson”, “Elton”, “Garry” };
Console.WriteLine(“Employee Names”);
foreach (string names in
employeeNames)
{
  Console.WriteLine(“{0} ”, names);
}
```

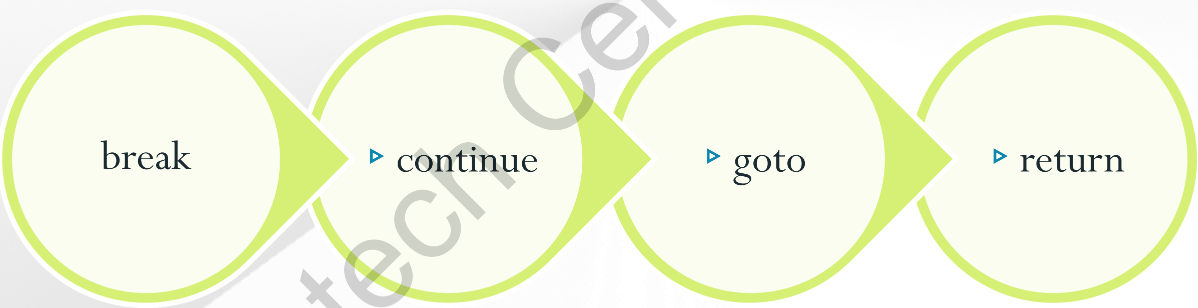


Output

```
Employee Names
Maria
Wilson
Elton
Garry
```

Jump Statements in C#

- Jump statements are used to transfer control from one point in a program to another.
- C# supports four types of jump statements.



break

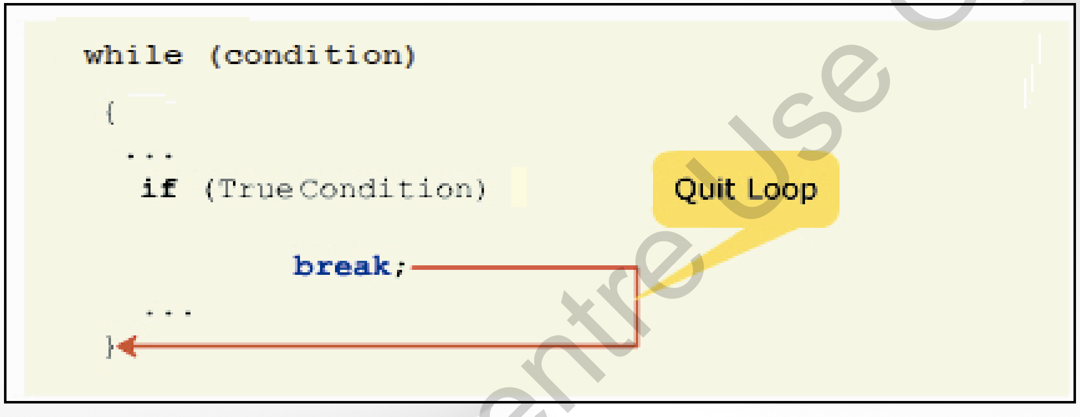
- continue
- goto
- return

The break Statement

- The break statement is used in the selection and loop constructs.

  It is most widely used in the switch…case construct and in the for and while loops.

Syntax



```
while (condition)
 {
  ...
   if (TrueCondition)
           break;
    ...
 }
```

Snippet

```
int numOne = 17;
int numTwo = 2;
while(numTwo <= numOne-1)
{
    if(numOne % numTwo == 0)
    {
      Console.WriteLine(“Not a Prime Number”);
      break;
    }
    numTwo++;
}
if(numTwo == numOne)
{
      Console.WriteLine(“Prime Number”);
}
```

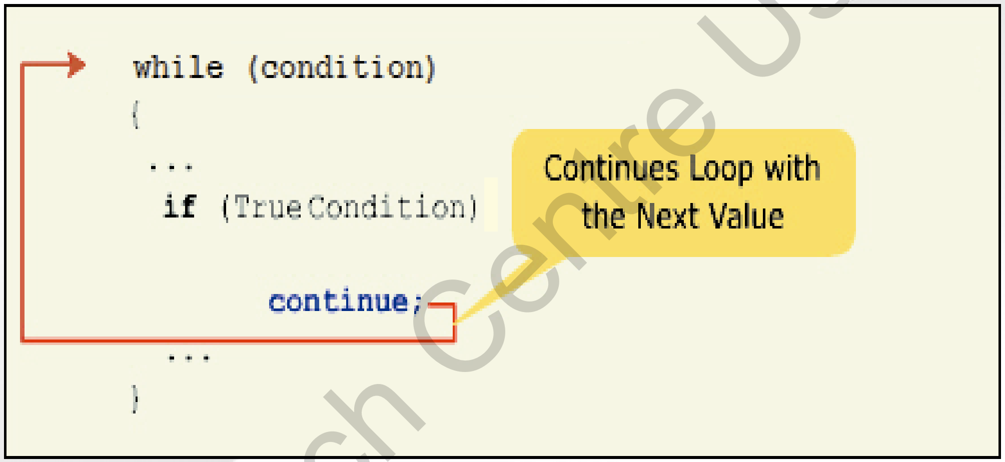
Output

```
Prime Number
```

The `continue` Statement

- The continue statement is most widely used in the loop constructs and is denoted by the continue keyword.
- The continue statement is used to end the current iteration of the loop and transfer the program control back to the beginning of the loop.

Syntax



```
while (condition)
{
 ...
  if (TrueCondition)

        continue;
  ...
}
```

Continues Loop with the Next Value

Snippet

```
Console.WriteLine("Even numbers in the range of 1-10");
for (int i=1; i<=10; i++)
{
  if (i % 2 != 0)
  {
        continue;
  }
Console.Write(i + " ");
}
```

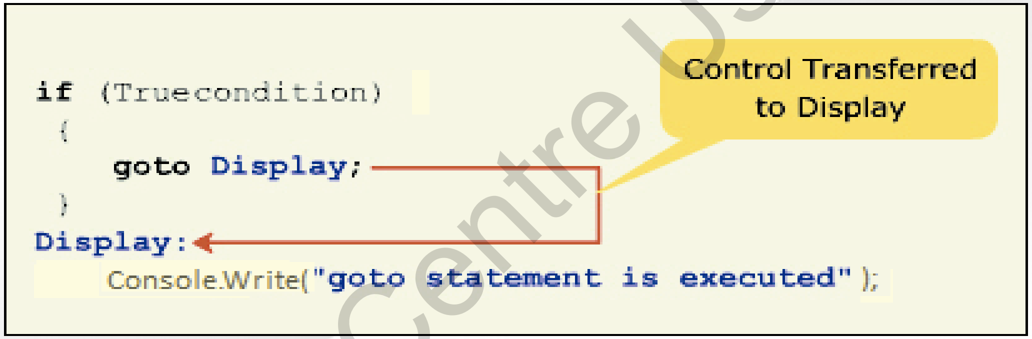
Output

```
Even numbers in the range of 1-10
2 4 6 8 10
```

The goto Statement

- The **goto** statement allows you to directly execute a labeled statement or a labeled block of statements.
- A labeled block or a labeled statement starts with a label. A label is an identifier ending with a colon.
- A single labeled block can be referred by more than one goto statements.
- The goto statement is denoted by the goto keyword.

Syntax



```
if (Truecondition)
 {
     goto Display;
 }
Display:
    Console.Write("goto statement is executed");
```

Control Transferred to Display

Snippet

```
int i = 0;
display:
Console.WriteLine("Hello World");
i++;
if (i < 5)
{
    goto display;
}
```

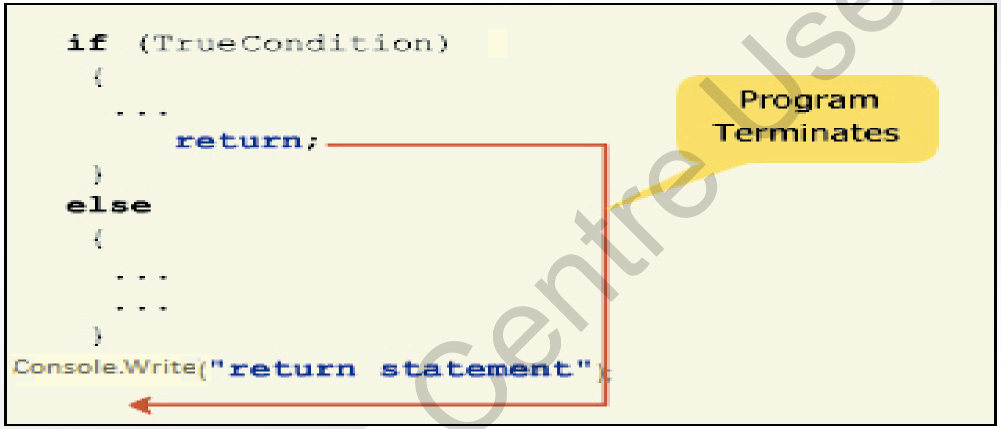
Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

The `return` Statement

- The return statement is used to return a value of an expression or is used to transfer the control to the method from which the currently executing method was invoked.

Syntax



```
  if (TrueCondition)
   {
    ...
       return;
   }
  else
   {
    ...
    ...
   }
Console.Write("return statement");
```

Program Terminates

Snippet

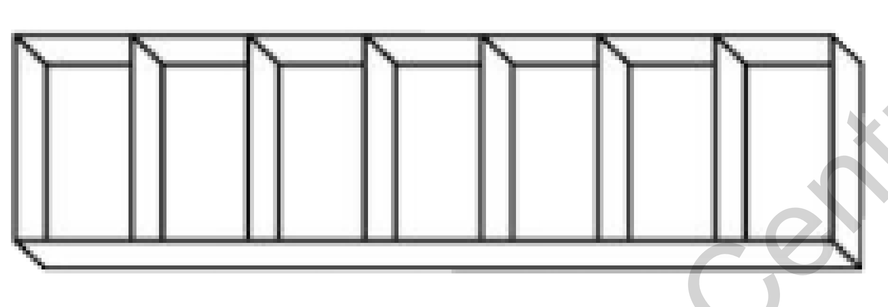
```
static void Main(string[] args)
{
    int num = 23;
    Console.WriteLine("Cube of {0} = {1}",num,Cube(num));
}
static int Cube(int n)
{
   return (n * n * n);
}
```

Output

```
Cube of 23 = 12167
```

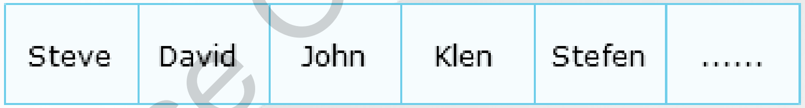
Introduction to Arrays

- An array is a collection of elements of a single data type stored in adjacent memory locations.



- In a program, an array can be defined to contain 100 elements to store the name of 100 students.

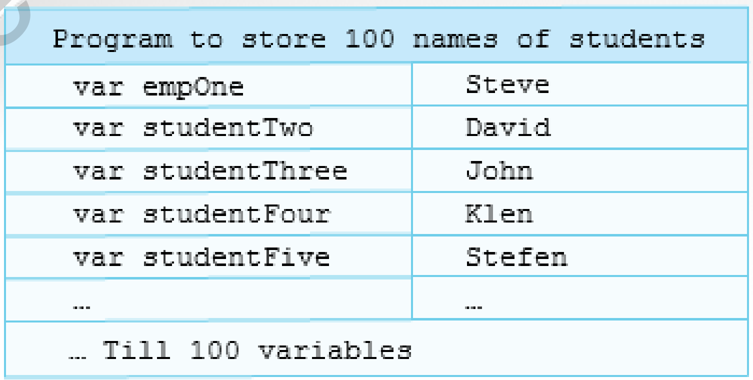
Array of 100 Names



Steve	David	John	Klen	Stefen	……

Proper Utilization of Memory

100 Variables Storing Names

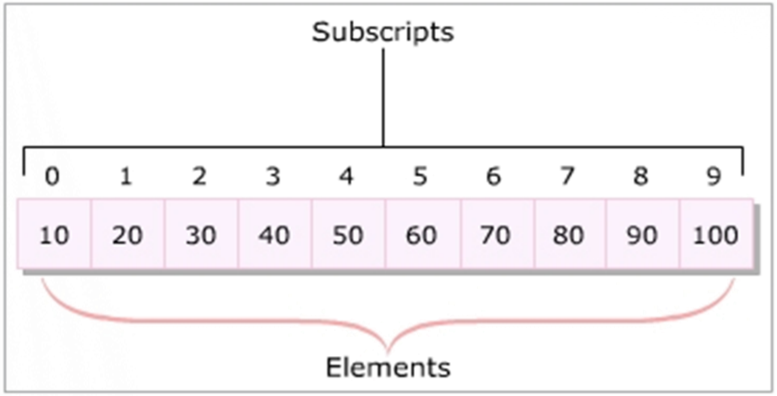


Program to store 100 names of students	
var empOne	Steve
var studentTwo	David
var studentThree	John
var studentFour	Klen
var studentFive	Stefen
…	…
… Till 100 variables	

Inefficient Memory Utilization

Definition

- An array always stores values of a single data type.
- Each value is referred to as an element.
- These elements are accessed using subscripts or index numbers that determine the position of the element in the array list.
- C# supports zero-based index values in an array.
- This means that the first array element has an index number zero while the last element has an index number n-1, where n stands for the total number of elements in the array.
- This arrangement of storing values helps in efficient storage of data, easy sorting of data, and easy tracking of the data length.



0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100

Arrays

Declaring Arrays

Arrays are reference type variables whose creation involves two steps:

- **Declaration:**
  - An array declaration specifies the type of data that it can hold and an identifier.
  - This identifier is basically an array name and is used with a subscript to retrieve or set the data value at that location.
- **Memory allocation:**
  - Declaring an array does not allocate memory to the array.

Syntax

```
type[] arrayName;
```

- In the syntax:
  - **type:** Specifies the data type of the array elements (for example, int and char).
  - **arrayName:** Specifies the name of the array.

Initializing Arrays 1-3

- An array can be:

  Created using the new keyword and then, initialized.

  Initialized at the time of declaration itself, in which case the new keyword is not used.

Data Types	Default Values
int	0
float	0.0
double	0.0
char	'\0'
string	Null

Syntax

- Syntax to declare and create an array in the same statement using the new keyword:

```
arrayName = new type[size-value];
```

Syntax

```
type[] arrayName = new type[size-value];
```

Initializing Arrays 2-3

Syntax

```
type[ ] arrayIdentifier = {val1, val2, val3, ..., valN};
```

- In the syntax:
  - **val1:** It is the value of the first element.
  - **valN:** It is the value of the nth element.
- Following code creates an integer array which can have a maximum of five elements in it:

Snippet

```
public int[] number = new int[5];
```

- Following code initializes an array of type string that assigns names at appropriate index locations:

```
public string[] studNames = new string{“Allan”, “Wilson”,
“James”, “Arnold”};
```

Initializing Arrays 3-3

- Following code stores the string 'Jack' as the name of the fifth enrolled student:

Snippet

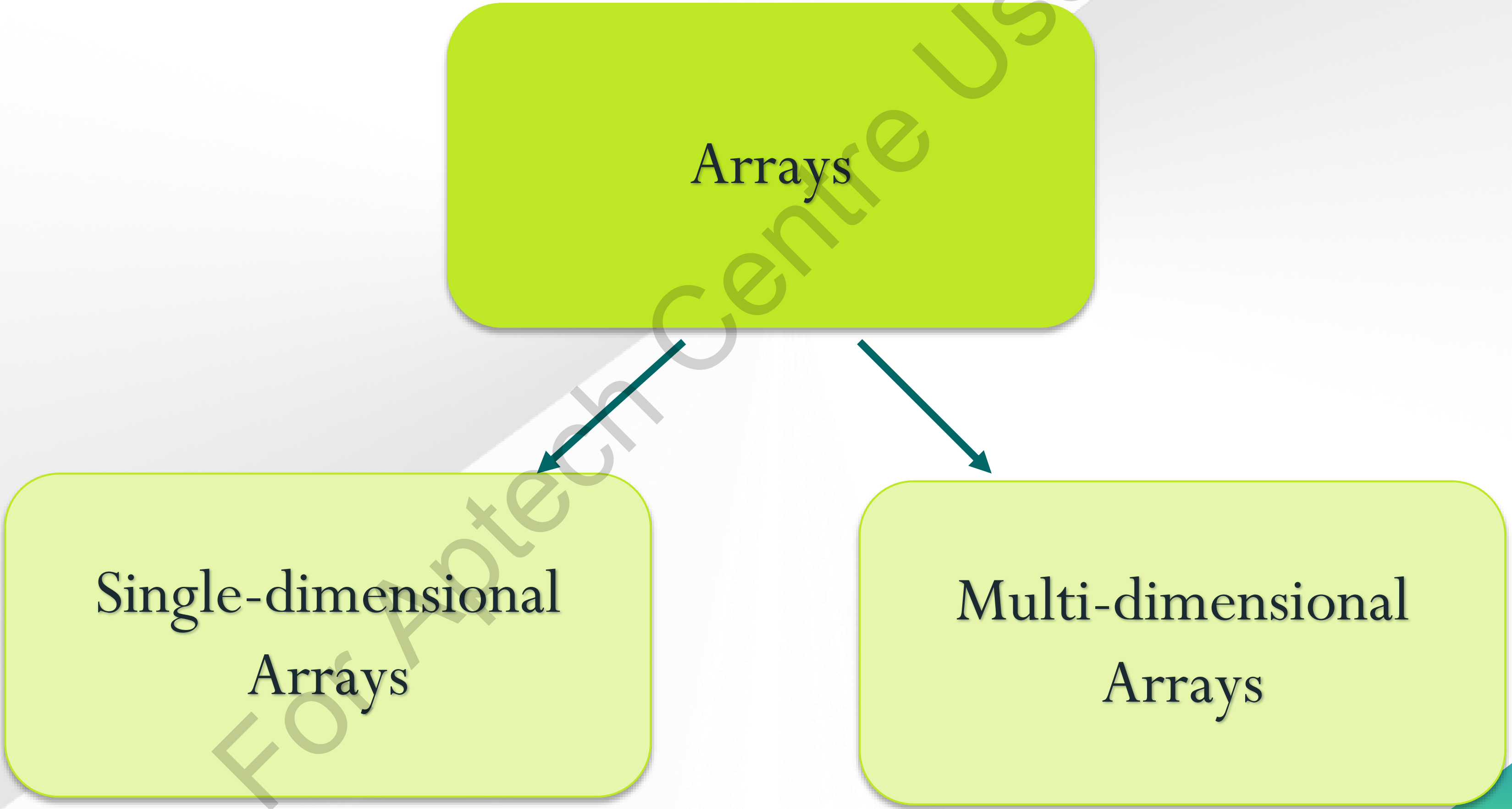
```
studNames[4] = “Jack”;
```

- Following code demonstrates another approach for creating and initializing an array. An array called count is created and is assigned int values:

```
using System;
class Numbers
{
   static void Main(string[] args)
   {
      int[] count = new int[10];//array is created
      int counter = 0;
      for(int i = 0; i< 10; i++)
      {
       count[i] = counter++; //values are assigned to the elements
      Console.WriteLine(“The count value is: ‟ + count[i]);
      //element values are printed
      }
     }
}
```

Types of Arrays

- Based on how arrays store elements, arrays can be categorized into following two types:



Arrays

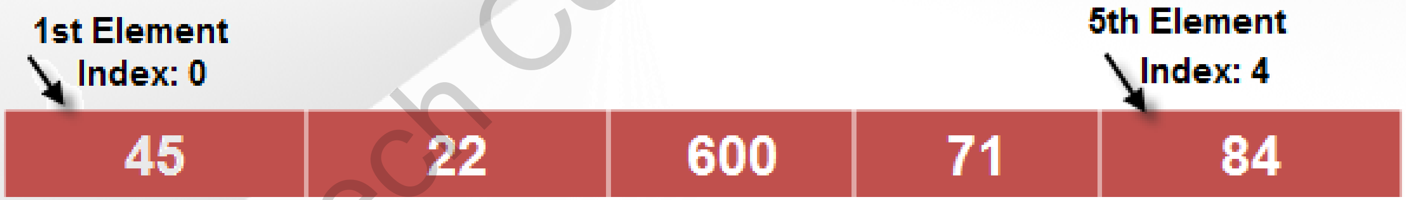
Single-dimensional Arrays

Multi-dimensional Arrays

Single-dimensional Arrays 1-2

- **Single-dimensional arrays:**
  - Elements of a single-dimensional array stored in a single row in allocated memory.
  - Declaration/initialization same as standard declaration/initialization of arrays.
  - Elements indexed from 0 to (n-1), where n is the total number of elements in the array.

Example



1st Element
Index: 0

5th Element
Index: 4

45	22	600	71	84

Syntax

```
type[] arrayName; //declaration
arrayName = new type[length]; // creation
```

Single-dimensional Arrays 2-2

- Following code initializes a single-dimensional array to store the name of students:

```
using System;

classSingleDimensionArray
{
static void Main(string[] args)
{
string[] students = new string[3] {“James”, “Alex”, “Fernando”};
for (int i=0; i<students.Length; i++)
{
Console.WriteLine(students[i]);
}
}
}
```

Output

```
James
Alex
Fernando
```

Multi-dimensional Arrays 1-2

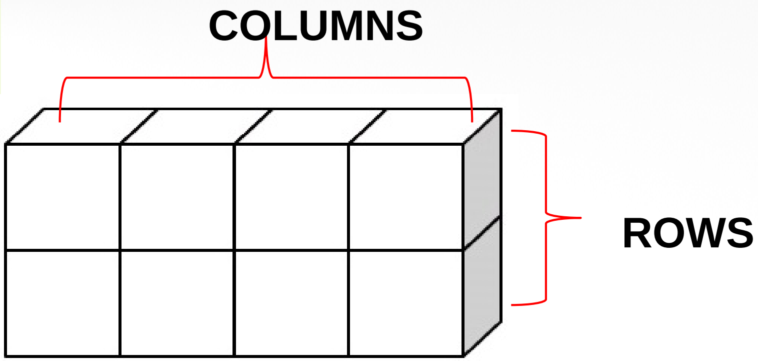
- A multi-dimensional array allows you to store combination of values of a single type in two or more dimensions.
- The dimensions of the array are represented as rows and columns similar to the rows and columns of a Microsoft Excel sheet.

Rectangular Array

- Is a multi-dimensional array where all the specified dimensions have constant values.
- Will always have the same number of columns for each row.

Jagged Array

- Is a multidimensional array where one of the specified dimensions can have varying sizes.
- Can have unequal number of columns for each row.



Multi-dimensional Arrays 2-2

Syntax

```
type[,] <arrayName>; //declaration
arrayName = new type[value1 , value2];
//initialization
```

Snippet

```
using System;
classRectangularArray
{
static void Main (string [] args)
{
int[,] dimension = new int [4, 5];
intnumOne = 0;
for (int i=0; i<4; i++)
{
for (int j=0; j<5; j++)
{
dimension [i, j] = numOne;
numOne++;
}
}
for (int i=0; i<4; i++)
{
for (int j=0; j<5; j++)
{
Console.Write(dimension [i, j] + “ ”);
}
Console.WriteLine();
}
}
}
```

Fixed and Dynamic Arrays 1-2

Fixed-length arrays

The number of elements is defined at the time of declaration.

For example, an array declared for storing days of the week will have exactly seven elements.

The number of elements is known and hence, can be defined at the time of declaration. Therefore, a fixed-length array can be used.

Dynamic arrays

The size of the array is not fixed at the time of the array declaration and can dynamically increase at runtime or whenever required.

For example, an array declared to store the e-mail addresses of all users who access a particular Web site cannot have a predefined length.

In such a case, the length of the array cannot be specified at the time of declaration and a dynamic array has to be used.

Can add more elements to the array as and when required.

Created using built-in classes of the .NET Framework.

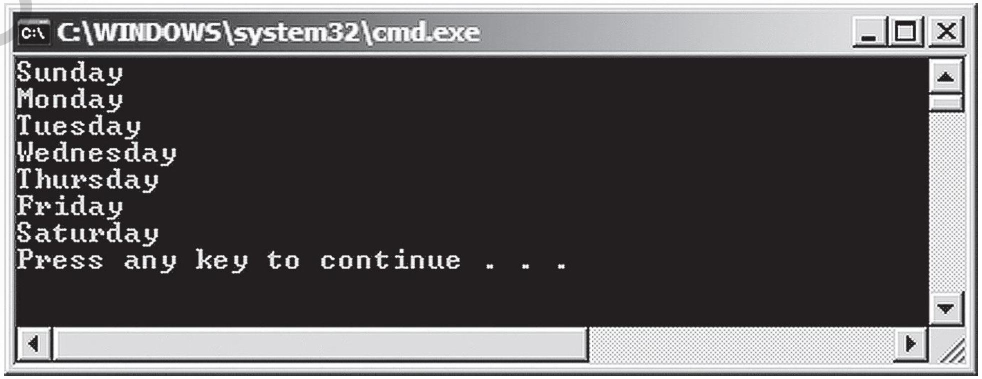
Fixed and Dynamic Arrays 2-2

- Code demonstrates fixed arrays.

Snippet

```
using System;
classDaysofWeek
{
static void Main(string[] args)
{
string[] days = new string[7];
days[0] = "Sunday";
days[1] = "Monday";
days[2] = "Tuesday";
days[3] = "Wednesday";
days[4] = "Thursday";
days[5] = "Friday";
days[6] = "Saturday";
for(int i = 0; i<days.Length; i++)
{
Console.WriteLine(days[i]);
}
}
}
```

Output



```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Press any key to continue . . .
```

Array References

- An array variable can be referenced by another array variable (referring variable).

```
using System;
classStudentReferences
{
    public static void Main()
    {
    string[] classOne = { "Allan", "Chris", "Monica" };
    string[] classTwo = { "Katie", "Niel", "Mark"};
    Console.WriteLine("Students of Class I:\tStudents of Class II");
    for (int i = 0; i< 3; i++)
    {
      Console.WriteLine(classOne[i] + "\t\t\t" + classTwo[i]);
    }

    classTwo = classOne;
    Console.WriteLine("\nStudents of Class II after referencing
    Class I:");
    for (int i = 0; i< 3; i++)
    {
      Console.WriteLine(classTwo[i] + " ");
    }
    Console.WriteLine();
    classTwo[2] = "Mike";
    Console.WriteLine("Students of Class I after changing the third
    student in Class II:");
    for (int i = 0; i< 3; i++)
    {
       Console.WriteLine(classOne[i] + " ");
    }
}
}
```

Rectangular Arrays 1-2

- A rectangular array is a two-dimensional array where each row has an equal number of columns.

Syntax

```
type [,]<variableName>;
variableName = new type[value1 , value2];
```

Snippet

```
using System;
class StudentsScore
{
   void StudentDetails()
   {
       Console.Write("Enter the number of Students: ");
       int noOfStds = Convert.ToInt32(Console.ReadLine());
       Console.Write("Enter the number of Exams: ");
       int exams = Convert.ToInt32(Console.ReadLine());
       string[] stdName = new string[noOfStds];
       string[,] details = new string[noOfStds, exams];

       for (int i = 0; i<noOfStds; i++)
       {
       Console.WriteLine();
       Console.Write("Enter the Student Name: ");
       stdName[i] = Convert.ToString(Console.ReadLine());
       for (int y = 0; y < exams; y++)
       {
       Console.Write("Enter Score in Exam " + (y + 1) + ": ");
       details[i, y] = Convert.ToString(Console.ReadLine());
```

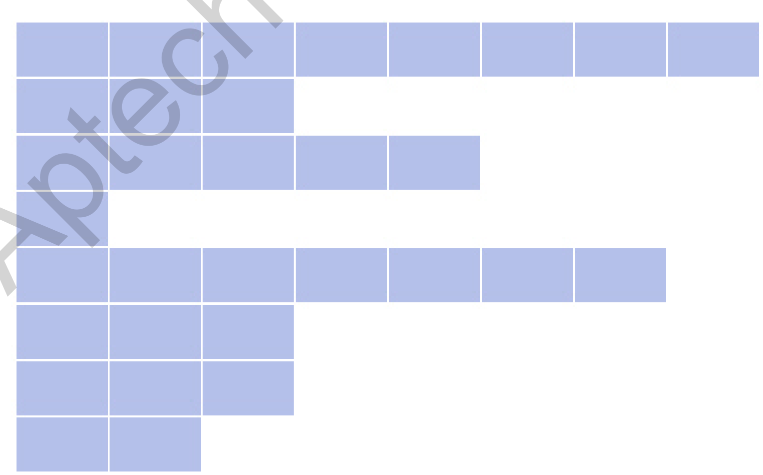
Rectangular Arrays 2-2

Snippet

```
      }
      }
      Console.WriteLine();
      Console.WriteLine("Student Exam Details");
      Console.WriteLine("--------------------");
      Console.WriteLine();
      Console.WriteLine("Student\t\tMarks");
      Console.WriteLine("-----\t\t------" );
      for (int i = 0; i<stdName.Length; i++)
      {
      Console.WriteLine(stdName[i]);
      for (int j = 0; j < exams; j++)
      {
      Console.WriteLine("\t\t" + details[i, j]);
      }
      Console.WriteLine();
   }
   }
   static void Main()
   {
   StudentsScore objStudentsScore = new StudentsScore();
   objStudentsScore.StudentDetails();
   }
}
```

Jagged Arrays 1-2

- A multi-dimensional array and is referred to as an array of arrays.
- Consists of multiple arrays where the number of elements within each array can be different. Thus, rows of jagged arrays can have different number of columns.
- Optimizes the memory utilization and performance because navigating and accessing elements in a jagged array is quicker as compared to other multi-dimensional arrays.



Jagged Arrays 2-2

Snippet

```
using System;

classJaggedArray
{
    static void Main (string[] args)
    {
      string[][] companies = new string[3][];
      companies[0] = new string[] {“Intel”, “AMD”};
      companies[1] = new string[] {“IBM”, “Microsoft”, “Sun”};
      companies[2] = new string[] {“HP”, “Canon”, “Lexmark”,
“Epson”};
      for (int i=0; i<companies.GetLength (0); i++)
      {
      Console.Write(“List of companies in group “ + (i+1) +
“:\t”);
      for (int j=0; j<companies[i].GetLength (0); j++)
      {
      Console.Write(companies [i][j] + “ “);
      }
      Console.WriteLine();
      }
    }

}
```

Output

```
List of companies in group 1: Intel AMD
List of companies in group 2: IBM Microsoft Sun
List of companies in group 3: HP Canon Lexmark Epson
```

Using the `foreach Loop` for Arrays

- Is used to perform specific actions on large data collections and can even be used on arrays.

Syntax

```
foreach(type<identifier> in <list>)
{
// statements
}
```

Snippet

```
using System

class Students
{
   static void Main(string[] args)
   {
    string[] studentNames = new string[3] { “Ashley”, “Joe”,
    “Mikel”};
    foreach (string studName in studentNames)
    {
      Console.WriteLine(“Congratulations!! “ + studName + “ you
       have been granted an extra leave”);
    }
   }
}
```

Array Class 1-2

- Is a built-in class in the System namespace and is the base class for all arrays in C#.
- Provides methods for various tasks such as creating, searching, copying, and sorting arrays.

Properties

- The Array class consists of system-defined properties and methods that are used to create and manipulate arrays in C#.
- The properties are also referred to as system array class properties.

Properties	Description
`IsFixedSize`	Returns a boolean value, which indicates whether the array has a fixed size or not. The default value is true.
`IsReadOnly`	Returns a boolean value, which indicates whether an array is read-only or not. The default value is false.
`IsSynchronized`	Returns a boolean value, which indicates whether an array can function well while being executed by multiple threads together. The default value is false.
`Length`	Returns a 32-bit integer value that denotes the total number of elements in an array.
`LongLength`	Returns a 64-bit integer value that denotes the total number of elements in an array.
`Rank`	Returns an integer value that denotes the rank, which is the number of dimensions in an array.
`SyncRoot`	Returns an object which is used to synchronize access to the array.

Array Class 2-2

- **Methods:**
  - The Array class allows you to clear, copy, search, and sort the elements declared in the array.
  - Following table displays the most commonly used methods in the Array class:

Methods	Descriptions
`Clear`	Deletes all elements within the array and sets the size of the array to 0.
`CopyTo`	Copies all elements of the current single-dimensional array to another single-dimensional array starting from the specified index position.
`GetLength`	Returns number of elements in an array.
`GetLowerBound`	Returns the lower bound of an array.
`GetUpperBound`	Returns the upper bound of an array.
`Initialize`	Initializes each element of the array by calling the default constructor of the Array class.
`Sort`	Sorts the elements in the single-dimensional array.
`SetValue`	Sets the specified value at the specified index position in the array.
`GetValue`	Gets the specified value from the specified index position in the array.

Using the `Array` Class 1-2

- The Array class allows you to create arrays using the **CreateInstance()** method.
- This method can be used with different parameters to create single-dimensional and multi-dimensional arrays.
- For creating an array using this class, you must invoke the **CreateInstance()** method that is accessed by specifying the class name because the method is declared as static.

Syntax for signature of the **CreateInstance()** method used for creating a single-dimensional array:

Syntax

```
public static Array CreateInstance(Type elementType, int length)
```

Syntax for signature of the **CreateInstance()** method used for creating a multi-dimensional array.

Syntax

```
public static Array CreateInstance(Type elementType, int length1, 
int length2)
```

Using the `Array` Class 2-2

Snippet

```
using System;

class Subjects
{
    static void Main(string [] args)
    {
      Array objArray = Array.CreateInstance(typeof (string), 5);
      objArray.SetValue(“Marketing”, 0);
      objArray.SetValue(“Finance”, 1);
      objArray.SetValue(“Human Resources”, 2);
      objArray.SetValue(“Information Technology”, 3);
      objArray.SetValue(“Business Administration”, 4);
      for (int i = 0; i<= objArray.GetUpperBound(0); i++)
      {
      Console.WriteLine(objArray.GetValue(i));
      }
    }
}
```

For manipulating an array, the Array class uses four interfaces

`ICloneable`

ICollection

IList

IEnumerable

Summary

- Selection constructs are decision-making blocks that execute a group of statements based on the boolean value of a condition.
- C# supports `if…else`, `if…else…if`, nested `if`, and `switch…case` selection constructs.
- Loop constructs execute a block of statement repeatedly for a particular condition.
- C# supports `while`, `do-while`, `for`, and `foreach` loop constructs.
- The loop control variables are often created for loops such as the for loop.
- Jump statements transfer the control to any labeled statement or block within a program.
- C# supports `break`, `continue`, `goto`, and `return` jump statements. Arrays are a collection of values of the same data type.
- C# supports zero-based index feature.
- There are two types of arrays in C#: Single-dimensional and Multi-dimensional arrays.
- A single-dimensional array stores values in a single row whereas a multi-dimensional array stores values in a combination of rows and columns.
- Multi-dimensional arrays can be further classified into rectangular and jagged arrays.
- The `Array` class defined in the `System` namespace enables to create arrays easily.
- The `Array` class contains the `CreateInstance`() method, which allows you to create single and multi-dimensional arrays.