C# PROGRAMMING

TEXT BOOK:

E. Balagurusamy, "Programming in C# :A Primer", Fourth Edition, McGraw Hill Education Private Limited, 2016.

Prepared by

B.Loganathan

C# PROGRAMMING

- **UNIT-I: Introduction to C#**: Evolution of C# – Characteristics of C# – How does C# differ from C++ and Java – Literals – Variables – Data types – Boxing and Un-boxing – Operators and Expressions- Arithmetic – Relational – Logical – Assignment – Increment and Decrement – Conditional – Bitwise and special operators – Type conversions – Mathematical functions – **Decision making and branching** : Decision making with if statement-Simple if statement-The if… else statement –Nesting of if…else statements- else…if Ladder- Switch statement- ?: Operator – Looping: While statement – do statement – for statement – for each statement – jumps in loops.

Introduction to C#:

- C# (pronounced as 'C sharp') is a new computer-programming language developed by Microsoft Corporation, USA.
- It has been designed to support the key features of *.NET Framework*, the new development platform of Microsoft for building component-based software solutions. It is a simple, efficient, productive and type-safe language.
- It is a purely objected-oriented, modern language suitable for developing *Web- based* applications.

Evolution of C#

- We have a number of limitations in using the WWW over the Internet.
  - We can see only one site at a time
  - The site has to be authored to our hardware environment.
  - The information we get is basically read-only
  - we cannot compare dynamically similar information stored in different sites
  - The Internet is a collection of many information islands that do not co-operate with each other. It continues to be a browsing and presentation network rather than an intelligent knowledge management network.

- Microsoft Chairman Bill Gates, the architect of many innovative and path-breaking softwa-re products during the past two decades, wanted to develop a *software platform* which will overcome these limitations.
- He wanted to make the Web both programmable and intelligent. T he outcome is a new generation platform called .NET and is simply the Microsoft's vision of software as a service.
- The research and developn1ent work of .NET platform began in the mid-90s, only during the Microsoft Professional Developers Conference in September 2000,was .NET officially announced to the developer community.

- At the same conference, Microsoft introduced C# as a de facto language of the .NET platform.
- Like Java, C# is a descendant of C++, which in turn is a descendant of C.
- C# both Java's features such as grouping of classes, interfaces and implementation together in one files that programmers can edit the code more easily.
- C# uses VB's approach to form design, namely, dragging controls from a tool box, dropping on to forms, and writing event handlers for them.

Characteristics of C#

- The main design goal of C# was simplicity rather than pure power. C# fulfills the need for a language that is easy to write, read and maintain and also provides the power and flexibility of C++.
- Main key features are :
  - Simple
  - Type-safe
  - Compatible
  - Consistent
  - Modern

  Object-oriented
  Versionable
  Interoperable
  Flexible and

- **i) Simple**
- C# simplifies C++ by eliminating some operators such as ->,:: and pointers. C# treats integer and Boolean data types as two entirely different types. This means that the use of= in place of == in **if** statements will be caught by the compiler.
- ii) **Consistent**
- C# supports an unified type system which eliminates the problem of varying ranges of integer types. All types are treated as objects and developers can extend the type system simply and easily.

iii) Modern

- C# is called a modem language due to a number of features it supports. It supports
  - Automatic garbage collection
  - Modern approach to debugging and
  - Rich intrinsic model for error handling
  - Robust security model
  - Decimal data type for financial applications

iv) Object-Oriented

- C# is truly object-oriented. It supports all the three tenets of object-oriented systems, namely,
- Encapsulation
- Inheritance
- Polymorphism

v) Type-safe

- Type-safety promotes robust programs. C# incorporates a number of type-safe measures.
    - All dynamically allocated objects and arrays are initialized to zero
    - Use of any uninitialized variables produces an error 1nessage by tile compile r
    - Access to arrays are range-checked and warned if it goes out-of-bounds
    - C# does not permit unsafe casts
  - C# enforces overflow checking in arithmetic operations
  - Reference parameters that are passed are type-safe
  - C# supports automatic garbage collection
-

vi) Versionable

- Making new versions of software modules work with the existing applications is known as *versioning.* C# provides support for versioning with the help of **new** and **override** keywords.

vii) Compatible

- C# enforces the .NET common language specifications and therefore allows inter-operation with other .NET languages.

viii) Flexible

- Although C# does not support pointers, we may declare certain classes and methods as 'unsafe‘ and then use pointers to manipulate them. However, these codes will not be type-safe.

ix) **Inter-operability**

- C# provides support for using COM objects, no matter what language was used to author them. C# also supports a special feature that enables a program to call out any native API.

- **<u>How does C# differ from C++</u>**
- The C# designers introduced a few changes in the syntax of C++ and removed a few features primarily to reduce the common pitfalls that occurred in C++ program development. They also added a number of additional features to make C# a type-safe and web-enabled language.

Changes Introduced

- C# compiles straight from source code to executable code, with no object files.
- C# does not separate class definition from implementation. Classes are defined and implemented in the same place and there is no need for header files.
- In C#, class definition does not use a semicolon at the end.
- The first character of the Main( ) function is capitalized. The Main must return either int or void type value.
- C# does not support #include statement. (Note that **using** is not the same as #include).

- All data types in C# are inherited from the object super class and there fore they are objects.
- All the basic value types will have the same size on any system. This is not the case in C or C++. Thus, C# is more suitable for writing distributed applications.
- In C#, data types belong to either value types (which are created in a stack) or reference types (which are created in a heap).
- C# checks for uninitialized variables and gives error messages at compile time. In C++, an uninitialized variable goes undetected thus resulting in unpredictable output.

◦ In C#, structs are value types.

- C++ *features dropped*
- The followingC++ features are mi ssing from C#:
- Macros
- Multiple inheritance
- Templates
- Pointers
- Global variables
- Typedef statement
- Default arguments
- Constant member functions or paramete rs
- Forward declaration of classes

How Does C# Differ From Java

- Like C#, Java was also derived from C++ and therefore they have similar roots. Moreover, C# was developed by Microsoft as an alternative to Java for web programming. C# has borrowed many good features from Java, which has already become a popular Internet language.
- Number of differences between C# and Java are:
  - Although C# uses .NET runtime that is similar to Java runtime, the C# compiler produces an executable code.

- C# has more primitive data types.
- Unlike Java, all C# data types are objects.
- Arrays *are* declared differently in C#.
- Although C# classes *are* quite similar to Java classes, there are a few important differences relating to constants, base classes and constructors, static constructors, versioning, accessibility of members etc.
- Java uses **static** final to declare a class constant while C# *uses* canst.
- *The* convention for Java is to put one public class in each file and in fact , some compilers require this. C# allows any source file arrangement.
- C# supports the struct type and Java does not.
- Java does not provide for operator overloading.

- **Literals**
- Literals are value constants assigned to variables (or results of expressions) in a program. C# supports several types of literals
- **Integer Literals :**
- An *integer* literal refers to a sequence of digits. There are two types of integers, namely, *decimal* integers and *hexadecimal* integers.
- Decimal integers consist of a set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer literals are:
- 123 –321 0 654321

- A sequence of digits preceded by Ox or OX is considered as a *hexadecimal* integer (hex integer). It may also include alphabets A through F or 'a ' through 'f‘.
- **Real Literals**
- Integer literals are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point)* number.
- A real literal may also be expressed in *exponential* (or *scientific) notation.* For example, the value 215.65 may be written as 2.I 565e2.

- **Boolean Literals**
- There are two Boolean literal values : True , False.
- They are used as values of relational expressions.
- **Single Character Literals**
- A single-character literal (or simply character constant) contains a single character enclosed within a pair of single quote marks. Example of character in the examples above constants are:
- **'5' 'X‘**

- **String Literals**
- A string literal is a sequence of characters enclosed between double quotes.
- The characters may be alphabets, dig its, special characters and blank spaces . Examples are:
- "Hello *C#* “ “2001" "WELLDONE" "?... !" "5+3" "X“
- **Backslash Character Literal**
- C# supports some special backslash character constants that are used in output methods. For example, the symbol ‘\n' stands for a new-line character.

- **Variables**
- A *variable* is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. Every variable has a type that determines what values can be stored in the variable.
- Some examples of variable names are:
  - average • total_height
  - height • classStreugth

Data type

Every variable in C# is associated with a data type. Data types specify the size and type of values that can be stored. C# is a language rich in its *data types.*

- The types in C# are primarily divided into two categories:
  - Value types
  - Reference types
- Value types and reference types differ in two characteristics:
  - Where they are stored in the memory
  - How they behave in the context of assignment statements

- Value types (which are of fixed length) are stored on the *stack.* Reference types (which are of variable length) are stored on the *heap.*

- The value types of C# can be grouped into *two* categories), namely,
  - User-defined types (or complex types) and
  - Predefined types (or simple types)
- We can define our own complex types known as *user-defined* value types which include **struct** types and enumerations.
- Predefined value types which are also known as *simple types* (or primitive types) are further subdivided into:
  - Numeric types,
  - Boolean types, and
  - Character types.

Boxing and Un-boxing

- In object-oriented progra1nming, methods are invoked using objects. Since value types such as **int** and **long** are not objects, we cannot use them to call methods.
- C# enables us to achieve this through a technique known as *boxing* . Boxing means the conversion of a value type on the stack to a object type on the heap. Conversely, the conversion from an object type back to a value type is known as *unboxing.*

- Consider the following code:
- int m = 10:
- object om = m;
- m = 20;
- Console.Writeline(m); */ /m=* 20
  Console.Writeline(om); // om = 10
- When a code changes the value of m, the value of om is not affected.
- Unboxing is the process of converting the object type back to the value type.
- Int m=10;
- object om = m; //box m
- int n = (int)om; //unbox om back to an int

Operators and Expressions

- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- C# supports a rich set of operators. C# operators can be classified into a number of related categories as below:
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Increment and decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators

- **Arithmetic Operators**
- C # provides all the basic arithmetic operators. They are:
- +   Add it ion or unary plus
- – Subtraction or unary minus
- * Multiplication
- /   Division
- %   Modulo division
- **Relational Operators**
- We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons or the price of two items and so on. These comparisons can be done with the help of *relational operators.*

- C# supports six relational operators:
- <   is less than
- <= is less than or equal to
- >   is greater than
- >= is greater than or equal to
- == is equal to
- != is not equal to
- **LOGICAL OPERATORS**
- C# has six logical operators :
- && logical AND
- II logical OR
- !   logical NOT
- &   bitwise logical AND
- | bitwise logical OR
- ^   bitwise logical exclusive OR

- **Assignment Operator**
- Assignment operators are used to assign the value of an expression to a variable.
- C# has a set of ‘shorthand' assignment operators which a re used in the form:
- v op = exp
- where v is a variable, *exp* is an expression and *op* is a C# binary operator. The operator **op** = is known as the *shorthand assignment operator.*
- Consider an example
- **X += y+1;**
- This is same as the statement
- X = X+(y+1);

- **Increment and Decrement Operators**
- These are the increment and decrement operators:
- ++ **and** – –
- The operator ++ adds 1 to the operand while –– subtracts 1.
- ++m; is equivalent to m = m + **1**; (or m + = 1;)
- – –m; is equivalent to m = m – 1; (or m – =**1**;)
- Consider the following:
- **m = 5;**
- **y = ++m;**
- In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as
- m = 5;
- **y = m++;**
- then, the value of y would be 5 and m would be 6

- **Conditional Operator**
- The character pair ? : is a *ternary opera tor* available in C#. This operator is used to construct conditional expressions of the form
- *exp1* ? *exp2* : *exp3*
- where *exp1, exp2* and *exp3* are expressions.
- *Exp1* is evaluated first. If it is true, then the expression *exp2* is evaluated and becomes the value of the conditional expression. If *exp1* is false , *exp3* is evaluated and its value becomes the value of the conditional expression.
- For example, consider the following statements:
- a= 10;
- b = 15;
- x = (a>b)? a : b;
- In this example, x will be assigned the value of b.

- **Bitwise and special operators**
- The bitwise logical and shift operators are :
- & bitwise logical AND
- | bitwise logical OR
- ^ bitwise logical XOR
- ~ one's complement
- << shift left
- >> shift right
- **SPECIAL OPERATORS**
- C# supports the following special operators.
- **is**   (relational operator)
- **as**   (relational operator)
- **typeof**   (type operator)
- **sizeo f**   (size operator)
- **new**(object creator)
- **.(dot)**   (member-access operator)
- **checked**   (overflow checking)
- **unchecked**(prevention of overflow checking)

TYPE CONVERSIONS

- We often encounter situations where there is a need to convert a data of one type to another before it is used in arithmetic operations or to store a value of one type into a variable of another type. For example, consider the code below:
- byte b1 = 50;
- byte b2 = 60;
- byte b3 = b1 + b2;
- This code attempts to add two byte values and to store the result into a third byte variable. But this will not work. The compiler will give an error message:
- "cannot implicitly convert type int to type byte."

- When we add two byte values, the compiler automatically converts then, into **int** types and the result of addition is an **int** value, not another **byte.**
- This is because the sum of two byte values may very easily result in a value that is much larger than the range of a **byte.**
- we may assign the result to an **int** type variable:
- int b3 = b1 + b2; / / no error
- Here, the compiler does the conversion for us, without out explicit request to do so.

- In C#, type conversions take place in two ways:
  - Implicit conversions
  - Explicit conversions
- **Implicit Conversions**
- Implicit conversions are those that will always succeed. That is, the conversion can always be performed without any loss of data.
- For example , a **short** can be converted implicitly to an **int,** because the **short** range is a subset of the **int** range. Therefore,
- short b = 75;
- int a = b, / / implicit conversion.
- are valid statements.

- Implicit conversions are possible in the following cases:
  - From **byte** to **decimal**
  - From **uint** to **double**
  - From **ushort** to **long**

Explicit Conversions

- There are many conversions that cannot be implicitly made between types. If we attempt such conversions, the compiler will give an error message. For example, the following conversions cannot be made implicitly:
  - **int** to **short**
  - **int** to **uint**
  - **uint** to **int**
  - **float** to **int**
  - **decimal** to any numeric type
  - e to **char**

- However, we can explicitly carry out such conversions using the 'cast' operator. The process is known as *casting* and is done as follows:
- *type* variable1 = (type) variable2;
- The destination type is placed **in** parentheses before the source variable.
- *Examples:*
- int m = 50;
- byte n = (byte) m;
- long x = 1234L;
- int y = (int) x;

MATHEMATICAL FUNCTIONS

- The **System** namespace defines a class known as **Math** class with a rich set of static methods that makes math-oriented programming easy and efficient.
- Some of the mathematical methods contained in the **Math** class:

Sin ()	sine of an angle in radians
Cos()	cosine of an angle in radians
Tan()	tangent of an angle in radians
Asin ()	inverse of sine
Acos ()	inverse of cosine
[redacted]	f the number

Atan ()	inverse of tangent
Atan2 ()	inverse tangent
Sinh ()	hyperbolic sine
Cosh ()	hyperbolic cosine
Tanh ()	hyperbolic tangent
Sqrt ()	square root
Pow()	number raised to a given power
Exp( )	exponential
Log ( )	natural logarithm (base e)
Log10()	base 10 logarithm
Abs( )	absolute value of a number
Min ()	lower of two numbers
Max()	higher of two numbers

- **Decision Making and Branching**
- A *C#* program is a set of statements that are normally executed sequentially in the order in which they appear. This happens when options or repetitions of certain calculations are not necessary. However, in practice , we have a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met.
- When a program breaks the sequential flow and Jump s to another part of the code, it is called *branching.* When the branching is based on a particular condition, it is known as *conditional branching.* If branching takes place without any decision, it is known as *unconditional branching.*
- C# language possesses such decision-ma king capabilities and supports the following statements known as *control* or *decision making* statements.

- DECISION MAKING WITH IF STATEMENT
- The **if** statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is a *two-way* decision statement and is used in conjunction with an expression. It takes the following form:
- *If(expression)*
- It allows the computer to evaluate the *expression* first and then, depending on whether the value of the *expression*(relation or condition) is 'true' or ' false' , it transfers the control to a particular statement.

- Some examples of decision making, usiJ1g the **if** statement are:
- **if** (bank balance is zero) borrow money
- **if** (room is dark) put on lights
- The **if** statement may be implemented in different forms depending on the complexity of the conditions to be tested.
- 1. Simple **if** statement 2.**if..else** statement
- 3. Nested **if..else** statement 4.**else if** ladder
- **SIMPL E IF STATEMENT**
- The general form of a simple **if** statement is if(boolean-expression)
- {
- statement-block;
- }
- statement-x;

- The 'statement-block' may be a single statement or a group of statements. If the *boolean-expression* is true, the *statement -block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x.*
- Consider the following segment of a program:
- if(category == SPORTS)
- {
- marks = marks + bonus_marks;
- }
- System.Console.WriteLine(marks);

- The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus_marks are added to his marks before they are printed. For others, bonus_marks are not added.
- **THE IF… ELSE STATEMENT**
- The **if ….else** statement is an extension of the simple **if** statement. The general form is :
- *If(boolean_expression)* {
- True-block statement(s)
- }
- else {
- False-block statement(s)
- }
-

- If the *boolean_expression* is true, then *the true-block statement(s),* immediately following the if statement, are executed; otherwise, the *false-block statement{(s)* are executed. In either case, either *true-block or false- block* will be executed, not both. In both the cases, the control is transferred subsequently to the statement-x.
- Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statements to do this maybe written as follows:

- if(code == 1)

      boy = boy +1;
- else

      girl= girl + 1;
- xxx;
- Here, if the code is equal to 1 , the statement **boy = boy + 1** ; is executed and the control is transferred to the statement **xxx,** after skipping the **else** part. If the code is not equal to 1, the statement **boy= boy** + **1**; is skipped and the statement in the else part **girl** = **girl**+**1**; is executed before the control reaches the statement **xxx.**

- **NESTING OF IF. .. .ELSE STATEMENTS**
- When a series of decisions are involved, we may have to use more than one if....else statement in *nested* form as follows:
- If (test condition1) {
  if(test condition2) {
-         statement1;
-     else {
-         statement2;
-         }
- else {
- satement3;
- }
- statement-x;
- *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, then statement-1 will be [illegible]ise statement-2 will be evaluated and the [illegible]erred to statement-x.

- A commercial bank has introduced an incentive policy of giving a bonus to all its deposit holders. The policy is as follows: A bonus of 2 percent of the *balance* held on 31st December is given to every one, irrespective of their balances, and *5* percent is given to female account holders if their balance is more than Rs 5000. This logic can be coded as follows:
- if(sex is female)
- {

```
  if (balance > 5000)
      bonus = 0.05 * balance ;
  else
      bonus = 0.02 * balance;
```

- }
- else
- {
- bonus = 0.02 * balance ;
- }
- balance = balance + bonus;
- When nesting, care should be exercised to match every **if** with an **else.**
- **THE ELSE..IF LADDER**
- There is another way of putting **if-else** together when multipath decisions are involved. A multipath decision is a chain of **if-else** in which the statement associated with each **else** is an **if.** It takes the l form:

-
  ```
  if (condition-1)
        statement-1;
  else if (condition-2)
        statement-2;
        else if (condition-3)
               statement-3;
               ...
         else if (condition-n)
               statement-n;
         else
               default-statement;
  statement-x;
  ```
- This construct is known as the **else if** ladder. The conditions are evaluated from the top (of the [redacted]wards.

- As soon as the true condition is found, the statement associated with it is executed and the control is transferred to the *statement-x* (skipping the rest of the ladder). When all the n conditions become false, then the final **else** containing the *default-statement* will be executed.
- Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

80 to 100	Honours
6010 79	First Division
5010 *59*	Second Division
40 to 49	Third Division
0 to 39	Fail

- This grading can be done using the else if ladder as follows:

```
if(marks > 79)
        grade = “Honours";
else if(marks > 59)
        grade = "First Division";
        else if(marks > 49)
                grade = "Second Di vision";
                else if(marks > 39)
                        grade ="Third Division";
                        else
                        grade= "Fail";
Console .WriteLine("Grade : "+ grade );
```

- **THE SWITCH STATEMENT**
- When one of many alternatives has to be selected, we can design a program using **if** state1nents to control the selection. However, the complexity of such a program increases dramatically when the alter natives increase .
- C# has a built-in multi-way decision statement known as a **switch.** The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed.

- The general form of the **switch** statement is as shown below:

```
switch(expression)
{
case value-1:
        block-1
        break;
case value-2:
        block-2
        break;
default:
         default-block
         break;
}
```

- The *expression* must be an integer type or **char** or **string** type. *value-1, value-2* are constants or constant expressions and are known as *case labels.* Each of these values should be unique within a **switch** statement.
- *block-1, block-2* .... are statement lists and may contain zero or more statements. There is no need to put braces around these blocks but it is important to note that case labels end with a colon (:).
- The **switch** statement is executed in the following order:

- The expression is evaluated first.
- The value of the expression is successively compared against the values, *value-1,value-2,…* If a case is found whose value matches the value of the *expression,* then the block of statements that follows the case are executed.
  - The break statement at the end of each block signals the end of a pa1ticular case and causes an exit from the **switch** statement, transferring the control to the *statement-x* following the **switch.**
  - The **default** is an optional case. When present, it will be executed if the value of the expression does not match any of the case values. **If** not present, no action takes place when all matches fail and the control goes to the *statement-x.*

- The **switch** statement can be used to grade the students as :
- index = marks/10;
- switch(index)
- {
- case 10:
- case 9:
- case 8:
- **grade = "Honours";**
- break;
- case 7:
- case 6:
- **grade = "First Division";**
- break;
- **case 5:**
- grade = "Second Di vision" ;
- break;

- case 4:
- grade = "Third Division";
- break;
- default:
- grade= "Fail";
- break;
- }
- Console.Writeline(grade);
- Note that we have used a conversion statement index = marks / 10; where, index is defined as an integer. This segment of the program illustrates two important features.
- First, it uses empty cases. The first three cases will execute the same statements **grade = "Honours";** Second, the default condition is used for all other cases where marks are less than 40.

- **THE** ? : **OPERATOR**
- C # has an unusual operator, useful for making two-way decisions; it is a combination of ? and : and takes three operands. This operator is popularly known *as the conditional operator.* The general form of use of the *conditional operator* is as follows:
- *conditional -expression? expression1 : 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 returnted.

- For example, the segment
- if (x < 0)
-         flag = 0;
- else
-         flag = 1;
- can be written as
- flag = (x<0) ? 0 : 1;
- **Looping**
- A computer is well suited to perform repetitive operations. It can do so tirelessly ten, hundred or even ten thousand times. Every computer language must have features that instruct a computer to perform such repetitive tasks. The process of repeatedly executing a block of [redacted]nown as *looping.*

- A looping process, in general, would include the following four steps:
  - Setting and Initialization of a counter.
  - Execution of the statements In the loop.
  - Test for a specified condition for execution of the loop.
  - Incrementing the counter.
- The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met with. The C# language provides for four constructs for performing loop operations. They are:
- The **while** statement
- The **do** statement
- The **for** statement
- The **foreach** statement

- **THE WHILE STATEMENT**
- The simplest of all the looping structures in C# is the **while** statement. The basic format of the **while** statement is :
- *initialization;*
- while(test condition) {
- Body of the loop
- }
- **while** is an *entry -controlled loop* statement. The *test condition* is evaluated and if the condition is true, then the body of the loop is executed.
- After execution of the body, the test condition is once again evaluated and if it is [illegible] is executed once again.

- This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.
- The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.
- Consider the following code segment:
- sum = 0;
- n = 1; //counter
- while(n <= 10)

- {
- sum = sum + n * n;
- n = n+1;   //incrementing the number
- }
- System.Console.Wri teLine(" Sum = " + sum);
- The body of the loop is executed 10 times for n = l, 2, ..., 10 each time adding the square of the value of n, which is incremented inside the loop.
- **THE DO STATEMENT**
- On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takes th

- *Initialiazat ion;*
- do
- {
- body of the loop
- }
- while (test *condition);*
- On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop , the *test condition* in the **while** statement is evaluated.
- If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the condition is true. Then the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

- Since the *test condition* is evaluated at the bottom of the loop, the **do while** construct provides an *exit-controlled loop* and therefore the body of the loop is always executed at least once.
- Consider an example:
- i =1;
- sum = 0;
- do
- {
- sum = sum + i;
- i = i + 2;
- }
- while(sum < 40 || i < 10); //semicolon here
- The loop will be executed as long as one of the two relations is true.

- **THE FOR STATEMENT**
- **for** is another *entry-controlled loop* that provides a more concise loop-control structure. The general form of the for loop is:
- **for** *(initialization* ; *test condition; increment)*
- {
- Body of the loop
- }
- The execution of the **for** statement is as follows:
- *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables i and count are known as loop-control variables.

- The value of the control variable is tested using the *test condition.* The test condition is a relational expression, such as i < 10, which determines when the loop will exit. If the condition is *true,* the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition.

- If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.
- Consider the following segment of a program:
- for (x = 0 ; x <= 9 ; x = x+1)
- {
- System.ConsoleWriteLline(x);
- }
- This **for** loop is executed ten times and prints the digits 0 to 9. The three sections enclosed within parentheses must be separated by semicolons.

Additional Features of the for Loop

- The **for** loop has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized in the **for** statement. The statements:
- p = 1;
- for (n=0; n<17; ++n) can be rewritten as
- for (p=1, n=0; n<17;++n)
- Notice that the initialization section bas two parts p = 1 and n = 0 separated by a *comma.*
- Like the initialization section, the increment section may also have more than one part. For example,
- for (n=1, m=50; n<=m; n=n+1, m=m-1)

Nesting of for Loops

- Nesting of loops, that is, one for statement within another for statement, is allowed in C#.
- for (i= 1; i<10; ++i) //Outer loop
- {
- for(j= 1;j< 5; ++j)
- {
  Inner loop
- }
- A program segment to print a multiplication table using for loops is show n below:
- for (row=1; row <= ROWMAX; ++row) {
- for (column = 1; column <= COLMAX; ++ column) {
- Y = row + column;
- System.Console.Write(" " + Y);
- }
- System.Console.WriteLine(" ");
- }

- **THE FOREACH STATEMENT**
- The foreach statement is similar to the for statement but implemented differently. It enables us to iterate the elements in arrays and collection classes such as List and HashTable. The general form of the foreach statement is:
- foreach *(type variable* in expression)
- {
- Body of the loop
- }
- The *type* and *variable* declare the *iteration* variable. During execution, the iteration variable represents the array element (or collection element in case of collections) for which an iteration is currently being performed.

- The *expression* must be an *array* or *collection* type and an explicit conversion must exist from the element type of the collection to the type of the iteration variable. *Example:*
- public static *void* Main (string [ ] args)
- {
- foreach (string s in args)
- {
- Console. WriteLine(s);
- }
- }
- This program segment displays the command [redacted].

```
using System;
class ForeachTest
{
public static void Main ( )
{
int[ ] arrayInt = {11 , 22, 33, 44 };
foreach ( int m in arrayInt )
{
Console.Write(“ "+ m);
}
Console.WriteLine( );
}
}
```

- Program segment will display the following output:

```
11 22 33 44
```

JUMPS IN LOOPS

- Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.
- For example, consider the case of searching for a particular name in a list containing, say, a hundred names.

- **Jumping Out of a Loop**
- An early exit from a loop can be accomplished by using the break and goto statements. We have already seen the use of the break in the switch statement. These statements can also be used within while, do or for loops for an early exit.
- When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.
- During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions.

- Like the **break** statement, C# supports another similar statement called the **continue** statement. However, unlike **break** which causes the loop to be terminated, the **continue** statement, as the name implies, causes the loop to continue with the next iteration after skipping any statements in between.

Labeled Jumps

- We have seen that the **break** will enable us to come out of only the nearest loop and the **continue** will enable us to restart the current loop. If we want to jump a set of nested loops or to continue a loop that is outside the current one, we may have to use the concept labeling and the goto statement.

- A label is any valid C# variable name ending with a colon. We can use labels anywhere in a program and use the goto statement to transfer the control to the statement marked by the label. For Example*:*
- public static void Main(String [ ] args)
- {
- if (args.Length == 0)
- **goto** end;
- Console. WriteLine(args. Length);
- end: / / Label name
- Console. WriteLine("end");
- }
- We can use a **goto** statement and a label to [illegible] out of a nested loop also.

- For example code below:
- for (int i = 0; i < 10; i++)
- {
- while (x < 100) {
- y = i + x;
- if (y > 500)
- **goto** out;
- }
- }
- **out:**
- Here, the label **out** is outside the **for** loop and the refer the statement.
- goto out;
- causes the execution to break out of both the