C# PROGRAMMING

TEXT BOOK:

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

- **UNIT-II: Object Oriented Programming In C#**: Methods- Declaring Methods – Main Methods – Invoking Methods – Nesting of Methods – Method Parameters – Pass by value – Pass by Reference – Output Parameters – Handling Arrays in C# : One-dimensional arrays – Creating an arrays – Two-dimensional arrays – Variable size arrays – System. Array class – **Manipulating strings**: Creating strings – String methods – Inserting string – Comparing strings – Finding substrings – Mutable strings- Arrays of strings –**Structures and enumerations**: Structures – Structs with methods – Nested Structs – Enumerations – Initialization – Base type – Enumerator type conversion.

Object Oriented Programming In C#

- In object-oriented programming, objects are used as building blocks in developing a program. They are t he runtime entities. They may represent a person, a pl ace, a bank account, a table of data or any item that the program handles.
- **Methods in C#**
- Objects encapsulate data, and code to manipulate that data. The code designed to work on the data is known as *methods* in C#. Methods give objects their behavioral characteristics. They are used not only to access and process data contained in the object but also to provide responses to any messages received from other objects.

- **DECLARING METHODS**
- Methods are declared inside the body of a class, normally after the declaration of data fields. The general form of a method declaration is:
- *Modifiers type method-name (formal-parameter-list)*

{
method _ body
}

- Method declaration has five parts:
  - Name of the method *(method-name)*
  - Type of value the method returns *(type)*
  - List of parameters *(formal-parameter-list)*
  - Body of the method
  - rs *(modifier)*

- The *method-name* is a valid C# identifier. The *type* specifies the type of value the method will return.
- This can be a simple data type such as **int** as well as any class type. If the method does not return anything, we specify a return type of **void.**
- The *formal-parameter-list* is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input.
- Examples are:
- int Fun (int m, float x, float y) //three parameters
- void Display ( ) //no parameters

- The *modifiers* specify keywords that decide the nature of accessibility and the mode of application of the method. A method can take one or more of the modifiers list are :
- new :The method bides an inherited method with the same signature
- public : The method can be accessed from anywhere, including outside the class
- protected :The method can be accessed from within the class to which it be longs, or a type derived from that class
- internal : The method can be accessed from within the same program
- private : The method can only be accessed from inside the class to which it belongs

- static : The method does not operate on a specific instance of the class
- virtual : The method can be overridden by a derived class
- abstract : A virtual method which defines the signature of the method, but doesn't provide an implementation
- override : The method overrides an inherited virtual or abstract method
- sealed : The method overrides an inherited virtual method, but cannot be overridden by any classes which inherit from this class. Must be used in conjunction with override
- extern : The method is implemented externally, in a different language

THE MAIN METHOD

- C# programs start execution at a method named **Main( )**. This method must be the static method of a class and must have either **int** or **void** as return type.
- public static int Main( )
- Or
- public static void Main( )
- The modifier public is used as this method must be called from outside the program. The **Main** can also have parameters which may receive values from the command line at the time of execution.
- public static int Main (string [ ) args)
- public static void Main (string[ ] args)

- **INVOKING METHODS**
- Once methods have been defined, they must be activated for operations. The process of activating a method is known as *invoking* or *calling.* The invoking is done using the dot operator as shown below:
- *objectname.methodname(actual -paramet er-li st);*
- Here, *objectctname* is the name of the object on which we are calling the method *methodname.* T he *actual-parameter-list* is a comma separated list of ‘actual values' (or expressions) that must match in type, order and number with the formal parameter list of the *methodname* declared in the class.

```
using System;
class Method    / /    class containing the method
{
//Define the Cube method
int Cube ( int x ) {
return ( x · x · x );
} }
//Client class to invoke the cube method
class MethodTest {
public static void Main( ) {
//Create object for invoking cube
Method M = new Method ( );
int y = M.Cube (5);    //Method call
Console. Writeli ne( y ); //Write the result
}
```

- A method named cube and is used to compute the cube of a number passed in as a parameter. This above program will display an output of 125.
- **NESTING OF METHODS**
- A method can be called using only its name by another method of the same class. This is known as *nesting of methods.*
- For example below program contain class **Nesting** defines two methods, namely **Largest** () and **Max** (). The method **Largest** () calls the method **Max** () to determine the largest of the two numbers and then displays the result.

```
using System;
class Nesting {
void Largest ( int m, int n ) {
int large = Max ( m , n );     / / Nesting
Console.Writeline( large);
}
int Max (int a, int b) {
int x = ( a >b) ? a:b;
return(x);
} }
class NestTesting {
Public static Main() {
Nesting next = new Nesting();
Next.Largest(100,200); //Method call
} }
```

- A method can call any number of methods. It is also possible for a called method to call another method. That is, **Method1** may call **Method2,** which in turn may call **Method3.** Here, the method **Main** calls **Largest** which in turn calls **Max.** This program will produce an output of 200 that is the largest of **two values.**
- **METHOD PARAMETERS**
- A method invocation creates a copy, specific to that invocation, of the formal parameters and local variables of that method. The actual argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the body of a method, formal parameters can be used like any other variables.

- The invocation involves not only passing the values into the method but also getting back the results from the method. For managing the process of passing values and getting back the results, C# employs four kinds of parameters.
  - Value parameters
  - Output parameters
  - Reference parameters
  - Parameter arrays

Value parameters are used for passing parameters into methods by *value.* On the other hand, reference parameters are used to pass parameters into methods by *reference.* Output parameters, as the name implies, arc used to pass results back from a method. Parameter arrays are used in a method definition to enable it to receive variable number of arguments when called.

PASS BY VALUE

- By default, method parameters are *passed by value.* That is, a parameter declared with no modifier is passed by value and is called a *value parameter.*
- When a method is invoked, the values of actual parameters are assigned to the corresponding formal parameters. The values of the value parameters can be changed within the method.
- The below program will produce the output
- x = 100
- When the method **change( )** is invoked, the value of **x** is assigned to **m** and a new location for **m** is created in the memory. Therefore, any change in **m** does not affect the value stored in the location **x**.

- Illustration of Passing by Value:
- using System;
- class PassByValue
- {   static void Change(int m)

```
        {          m=m+1; // value of m is changed
        }
    public static void Main()
    {              int  x = 100;
                   Change(x);
                   Console.WriteLine("x="+x);
    } }
```

PASS BY REFERENCE

We can force the value parameters to be passed by reference. To do this, we use the ref key word. A parameter declared with the **ref** modifier is a reference parameter.

- *For Example:*
- void Modify *(**ref*** int x)
- Here, **x** is declared as a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, it represents the same storage location as the actual parameter used in the method invocation.
- Remember, when a formal parameter is declared as **ref** , the corresponding argument in the method invocation must also be declared as **ref**. *Example:*
- void Modify (ref int x)
- {
  X += 10; // value of m will be changed
- }
- int m = 5; // m is initialized
- Modify (ref m); // pass by *reference*

- Note the use of ***ref int*** as the parameter type in the definition and the use of ref keyword in the method call to tell the compiler *to pass by reference* rather than by value.
- Reference parameters a re used in situations where we would like to change the values of variable s in the calling method.
- **THE OUTPUT PARAMETERS**
- The *output* parameters are used to pass results back to the calling method. This is achieved by declaring the parameters with an **out** keyword. Similar to a reference parameter, an output parameter does not create a new storage location.

- When a formal parameter is declared as **out,** the corresponding actual parameter in the calling method must also be declared as **out.** For example,
- void Output (out int *x*)
- {
- x= 100;
- }
- int m; //m is uninitialized
- Output (out m); // value of m is set
- Note that the actual parameter **m** is not assigned any values before it is passed as output parameter. Since the parameters **x** and **m** refer to the same storage location, **m** takes the value that is assigned to **x.**
- Note that every formal output parameter of a method must be definitely assigned a value before the method returns. The following illustration is a simple [illegible]n of **out** parameters.

- using System;
- class Output
- { static void Square (int x, out int y)
- { y = x * x;

```
 }
 public static void Main() {
int m; // need not be initialized
Square(10, out m);
Console.WritLine(“m=“+m);
} }
```

Output of this Program would be: m = 100

Handling Arrays

- An array is a group of contiguous or related data items that share a common name. For instance, we can define an array name marks to represent a set of marks of a class of students. A particular value is indicated by writing a number called *index* number or *subscript* in brackets after t he array name. For example,
- marks[10]
- represents the marks obtained by the 10 student.

- **ONE-DIMENSIONAL ARRAYS**
- A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted* variable or a *one-dimensional* array.
- The subscripted variable xi, refers to the i-th element of x. In C#, a single-subscripted variable x, can be expressed as
- x[1], x[2], x[3]…x[n]
- The subscript can begin with number 0.
- **CREATING AN ARRAY**
- Like other variables, arrays must be declared and created in the compute r memory before they are used.

Creation of an array involves three steps:

- Declaring the array
  - Creating memory locations
  - Putting values into the memory locations.
- **Declaration of Arrays**
- Arrays in C# are declared as follows:
- *Type[]* arrayname;
- *Examples:*
- int[ ] counter; //declare int array reference
- float[ ] marks; //declare float array reference
- int[ ] x,y; //declare two int array reference
- Remember, we do not enter the size of the arrays in the declaration.
- **Creation of Arrays**
- After declaring an array, we need to create it in the memory. C# allows us to create arrays using **new** operator only, as shown below:
- *arrayname* • new type(size];

- *Examples:*
- number · new int[5];
- //create a 5 element *int* array
- average · new float[10];

//create a 10 element *float* array

These lines create the necessary memory locations for the arrays number and average and designate them as int and float respectively.

- **Initialization of Arrays**
- The final step is to put values into the an ay created. This process is known as *initialization.* This is done using the array subscripts as shown below.
- arrayname[subscript] = value ;

- *Example:*
- number[0] = 35; number (1) = 40;
- … number [4] = 19;
- Note that C# creates arrays starting with a subscript of 0 and ends with a value one less than the *size* specified.
- **TWO-DIMENSIONAL ARRAYS**
- There will be situations where a table of values will have to be stored.
- In mathematics, we represent a particular value in a matrix by using two subscripts such as Vij. Here, V denotes the entire matrix and ij refers to the value in the i-th row and j-th column.
- C# allows us to define such tables of items by using *two dimensional* arrays. The table can be re presented in C# as
- v[4,3]

- For creating two-dimensional arrays, we must follow the same steps as that of simple arrays. We may create a two-dimensional array like this:
- int[ ] myArray;
- myArray = new int[3,4];
- This creates a table that can store twelve integer values, four across and three down.
- Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,
- int[ ] table ={{0,0 ,0},{1,1,1}};

VARIABLE-SIZE ARRAYS

- C# treats multidimensional arrays as 'arrays of arrays'. It is possible to declare a two-dimensional array as follows:
- int[ ][] x= new int[3][ ]; // three rows array
- X[0] = new int[2];
- // first row has two elements
- x[1] = new int[4];
- // second row has four elements
- x[2] = new int[3];
- // third row has three elements
- These statements create a two-dimensional array having different lengths for each row.

- Variable-size arrays are called *jagged* arrays .
- The elements can be accessed as follows :
- X [1] [1] = 10 ;
- Int y = x[2[[2];
- Note the difference in the way we access the two types of arrays. With rectangular arrays, all indices are within one set of square brackets, while for jagged arrays each element is within its own square brackets.
- **THE SYSTEM ARRAY CLASS**
- In C#, every array we create is automatically derived from the **Systm.Array** class. This class defines a number of methods and properties that can be used to manipulate arrays more efficiently.

- Some of the commonly used System Array Class methods and their purpose are :
- Clear () :Sets a range of elements to empty values
- CopyTo ( ) :Copies elements from the source array into the destination array
- GetLength ( ) : Gives the number of elements in a given dimension of the array
- GetValue( ) : Gets the value for a given index in the array
- Length :Gives the length of an array
- SetValue ( ) :Sets the value for a given index in the array
- Reverse () :Reverses the contents of a one-dimensional array
- Sort () : Sorts the elements in a one-dimensional array

Manipulating Strings

- String manipulation is the most common part of many *C#* programs. Strings represent a sequence of characters.
- C# supports two types of strings, namely, *immutable* strings and *mutable.*
- C# also supports a feature known as *regular expressions* that can be used for complex string manipulations and pattern matching.
- **CREATING STRINGS**
- C# supports a predefined reference type known as **string.** We can use **string** to declare **string** type objects.

- We can create immutable strings using string or String objects in a number of ways.
- Assigning string literals
- Copying from one object to another
- Concatenating two objects
- Reading from the keyboard
- Using **ToString** method
- **Assigning String Literals**
- The most common way to create a string is to assign a quoted string of characters known as *string literal* to a string object. For example:
- string s1; //declaring a string object
- s1 = "abc“; //assigning string literal

Copying Strings

- We can also create new copies of existing strings. This can be accomplished in two ways:
  - Using the overloaded = operator
  - Using the static **Copy** method
- *Example:*
- string s2 = s1;   // assigning
- string s2 = string.Copy(s1);   //copying
- Both these statements would accomplish the same thing, namely, copying the contents of **s1** into **s2.**

Concatenating Strings

- We may also create new strings by concatenating existing strings. There are a couple of ways to accomplish this.
  - Using the overloaded + operator
  - Using the static **Concat** method
- *Examples:*
- string s3 = s1+ s2; //s1 & s2 exist already
  string s3 = string.Concat(s1, s2)
- **Reading from the Keyboard**
- It is possible to read a string value interactively from the keyboard and assign it to a string object.
- string s= Console. ReadLine( );

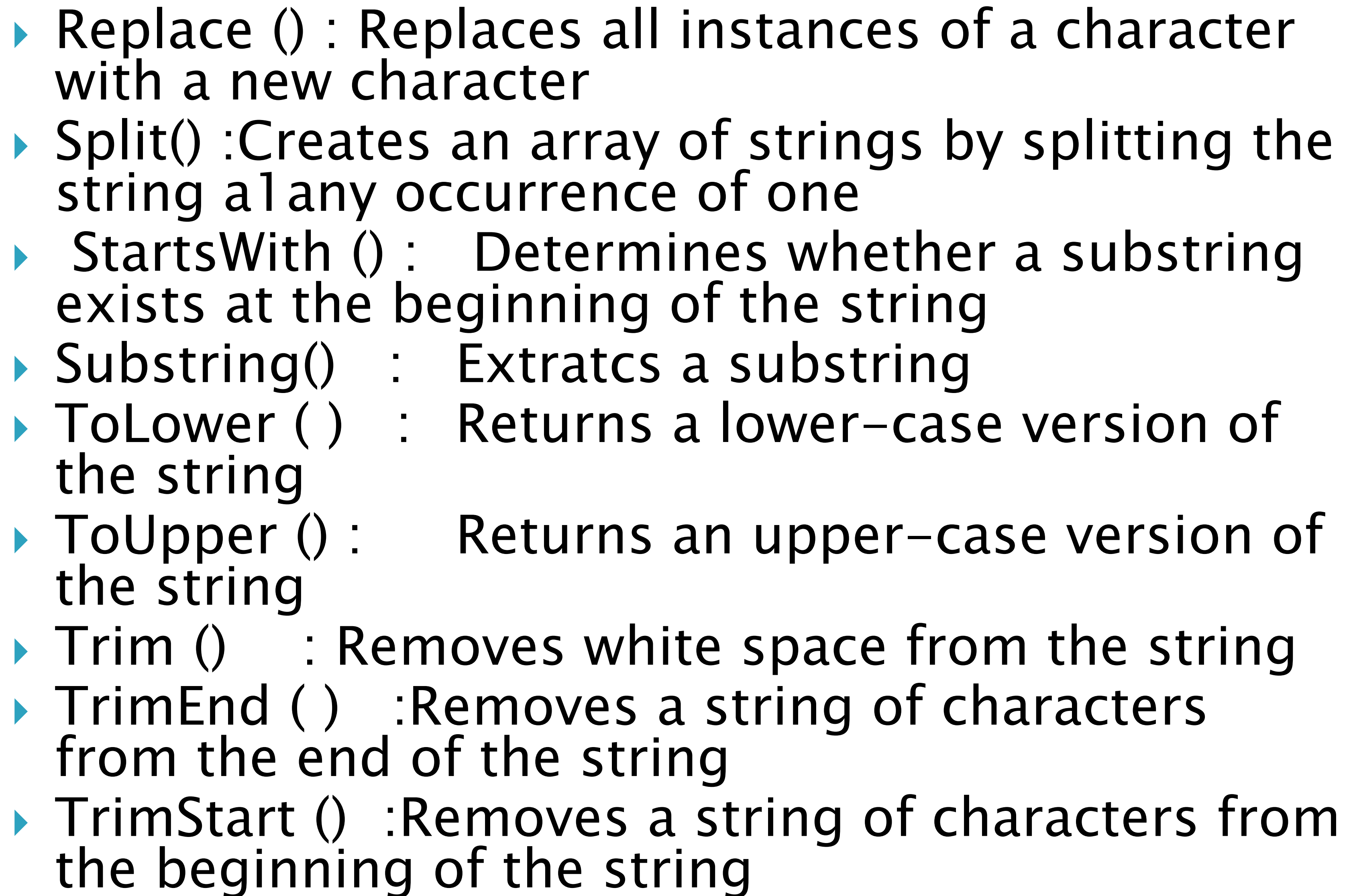
- On reaching this statement, the computer will wait for a string of characters to be entered from the keyboard. When the 'return key' is pressed, the string will be read and assigned to the string objects.

- **The ToString Method**
- Another way of creating a string is to call the **ToString** method on an object and assign the result 10 a string variable.
- int number = 123;
- string numStr = number.ToString();
- This statement converts the number 123 to a string ‘123' and then assigns the string value to the string variable **numStr.**

STRING METHODS

- **String** objects are *immutable,* meaning that we cannot modify the characters contained in them. However, since the string is an alias for the predefined **System.String** class in the Common Language Runtime (CLR).
- There are many built-in operations available that work with strings. The following list of various methods that could be used for various operations.
- Compare( ) :Compares two strings
- CompareTo() :Compares the current instance with another instance
- ConCat ( ) :Concatenates two or more strings
- Copy() :Creates a new String by copying

- CopyTo( ) :Copies a specified number of characters to an array of Unicode characters
- EndsWith () :Determines whether a substring exists at the end of the string
- Equals() :Determines if two strings are equal
- IndexOf() :Returns the position of the first occurrence of a substring
- Insert () :Returns a new string with a substring inserted at a specified location
- Join ( ) :Joins an array of strings together
- LastIndexOf ( ) :Returns the position of the last occurrence of a substring
- PadLeft ( ) :Left-aligns the strings in a field
- PadRight() :Right-aligns the string in a field
- Remove ( ) :Deletes characters from the string


Replace () : Replaces all instances of a character with a new character
Split() :Creates an array of strings by splitting the string a1any occurrence of one
StartsWith () : Determines whether a substring exists at the beginning of the string
Substring() : Extratcs a substring
ToLower ( ) : Returns a lower-case version of the string
ToUpper () : Returns an upper-case version of the string
Trim () : Removes white space from the string
TrimEnd ( ) :Removes a string of characters from the end of the string
TrimStart () :Removes a string of characters from the beginning of the string

- **INSERTING STRINGS**
- String methods are called using the string object on which we want to work. The following program illustrates the use of the **Insert** ( ) method and the indexer property supported by the **Systm.String** class.
- using system;
- class StringMethod
- {

```
    public static void Main()
    {
        string s1 = “Lean”;
        string s2 = s1.Insert (3,”r”);
                    s2.Insert (5,”er”);
```

- for (int I = 0; I <s3.Length;i++)
- Console.Write(s3[i]);
- Console.Write( );
- }
- When the statement
- string s2 = s1.Insert (3, "r");
- is executed, the string variable s2 contains the string "Learn". The string "r" is inserted in s1 after 3 characters. Similarly, the string "er" is inserted at the end of the string. Finally, the variable s3 contains the value "Learner".
- Note that, we are not modifying the contents of a given string variable. Rather, we are assigning the modified value to a new string

- **COMPARING STRINGS**
- String class supports overloaded methods and operators to compare whether two strings are equal or not. They are:
  - Overloaded **Compare()** method
  - Overloaded **Equals()** method
  - Overloaded == operator

i) Compare() Method

- There are two versions of overloaded static **Compare** method. The first one takes two strings as parameters and compares them. *Example:*
- int n string.Compare (s1,s2);

- This performs a case-sensitive comparison and returns different integer values for different conditions as under:
  - Zero integer, if s1 is equal to s2
  - A positive integer (1), if s1 is greater than s2
  - A negative integer (-1), if s1 is less than s2

  For example, if s1 = "abc" and s2 = "ABC", then n will be assigned a value of - 1. Remember, a lowercase letter has a smaller ASCII value than an uppercase letter.
- We can use such comparison statements in if statements like:
- if (string.Compare (s1, s2) == 0)
- Console. WriteLine("They are equal");

ii) Equals() Method

- The **string** class supports an overloaded **Equals** method for testing the equality of strings. There are again two versions of **Equals** method. They are implemented as follows:
- bool b1 = s2.Equals(s1);
- bool b2 = string.Equals (s2, s1);
- These methods return a Boolean value **true** if s1 and s2 are equal, otherwise **false.**
- **iii) The == Operator**
- A simple and natural way of testing the equality of strings is by using the overloaded = = operator.

- *For Example:*
- bool b3 = (s1 == s2); //b3 is true if they are equal
- We very often use such statements in decision statements, like:
- if (s1 == s2)
- Console.WriteLine("They are qual");
- **FINDING SUBSTRINGS**
- It is possible to extract substrings from a given string using the overloaded **Substring** method available in **String** class. There are two version of **Substring**:
  - s.Substring(n)
  - s.Substring(n1, n2)

- The first one extracts a substring starting from the nth position lo the last character of the string contained in s. The second one extracts a substring from s beginning at n1 position and ending at n2 position.
- *Examples:*
- string s1 = "NEW YORK”;
- string s2 = s1.Substring(5);
- string s3 = s1.Substring(0,3);
- string s4 = s1.Substring(5,8);
- When executed, the string variables will contain the following substrings:
- s2: YORK
- s3: NEW
- s4: YORK

- **MUTABLE STRINGS**
- Mutable strings that are modifiable can be created using the **StringBuilder** class. *Examples:*
- StringBuilder str1 = new StringBuilder("abc");
  StringBuilder str2 = new StringBuilder ( );
- The string object **str1** is created with an initial size of three characters and **str2** is created as an empty string. They can grow dynamically as more characters are added to them. They can grow either unbounded or up to a configurable maximum. Mutable strings are also known as *dynamic strings.*

- The **StringBuilder** class supports many methods that are useful for manipulating dynamic strings.
- Append () :Appends a string
- AppendFormat ( ) :Append strings using a specific format
- EnsureCapacity() :Ensures sufficient size
- Insert () :Insert a string at a specified position
- Remove () :Removes the specified characters
- Replace () :Replaces all instances of a character with a specified one.

- C# also supports some special functions known as *properties* . They are :
- Capacity :To retrieve or set the number of characters the object can hold
- Length :To retrieve or set the length
- MaxCapacity : To retrieve the maximum capacity of the object
- [ ] : To get or set a character at a specified position
- The **System.Text** namespace contains the **StringBuilder** class and therefore we must include the **using System.Text** directive for creating and manipulating mutable strings.

- using System.Text;   / / For using StringBuilder
- using System;
- class StringBuilderMethod
- {
- public static *void* Main( ) {
- StringBuilder s = new StringBuilder   (“Object ”);
- Console.WriteLine("Original string :“+s);
- Console.WriteLi ne(" Length :”+s.Length);
- / / Appending a string
- s.Append("language ");
- Console.Writeline("String now :”+s);
- / /Inserting a string
- s.Insert (7, "oriented ");
- Console.WriteLine("Modified string :”+s);
- } }

- Look at the output produced by Program :
- Original string Object
- Length 7
- String now Object language
- Modified string Object oriented language
- The above program accepts two string inputs from users and appends the first string to a predefined value. Using **StringBuilder** class, a string value is inserted in the string.
- **ARRAYS OF STRINGS**
- We can also create and use arrays that contain strings. The statement
- string [ ] itemArray = new string [3];
- will create an itemArray of size 3 to hold three strings

- We can assign the strings to the itemArray element by element using three different statements, or mo re efficiently using a for loop. We could also provide an array with a list of initial values in curly braces:
- string [] itemArra y = {"Java", "C++", "Csharp“};
- The size of the array is determined by the number of elements in the initialization list. The size of the array, once created, cannot be changed.
- If we want an array whose length is determined dynamically or an array which can be extended at run time, we have to use the **ArrayList** class to create a list.

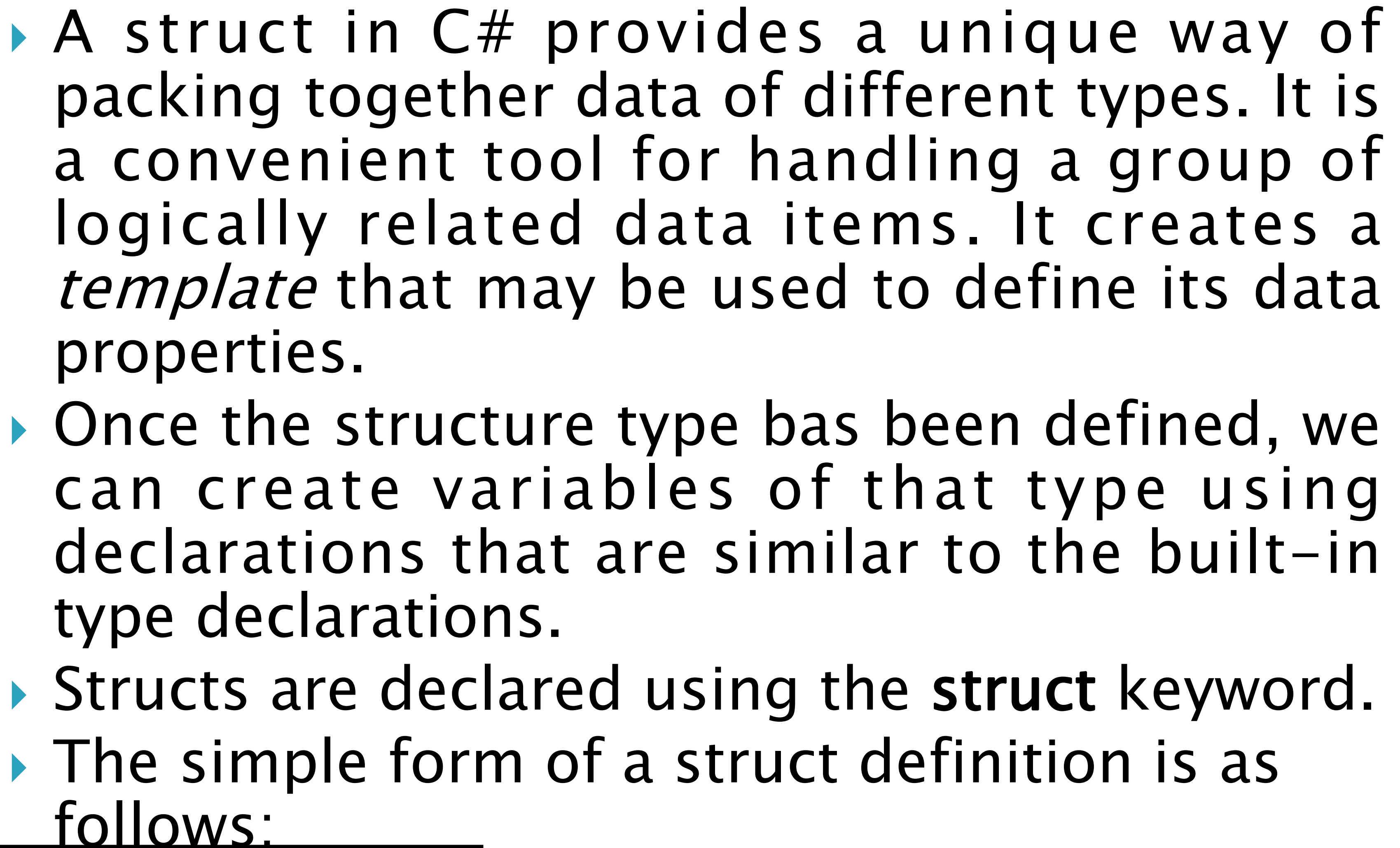
```
using System;
class Strings
{
    public static void Main()
    {
        string [ ]countries -=
  { “India“, ”Germany”, ”America”,  “France” };
Int n = countries.Length;
//Sort alphabetically
Array.Sort(countries);
for(int i=0; i <n; i++)   {
     Console.WriteLine(countries[i]);
}   }  }
```

Once an array of strings is created, we can sort them into ascending order or reverse their order using … Array class.

Structures and Enumerations

- C# allows us to define our own complex value types (known as user- defined value types) based on these simple data types. There are two sorts of value types we can define in C#:
- Structures
- Enumerations
- As we know, value type variables store their data on the stack and there fore the structures and enumerations are stored on the stack.

- **STRUCTURES**
- Structures (often referred to as *structs)* are similar to classes in C#. Although classes will be used to implement most objects, it is desirable to use structs where simple composite data types are required.
- Because they are value types stored on the stack, they have the following advantages compared to class objects stored on the heap:
- They are created much more quickly than heap-allocated types.
- They are instantly and automatically deallocated once they go out of scope.
- It is easy to copy value type variables on the stack.



- **Defining a Struct**
- A struct in C# provides a unique way of packing together data of different types. It is a convenient tool for handling a group of logically related data items. It creates a *template* that may be used to define its data properties.
- Once the structure type bas been defined, we can create variables of that type using declarations that are similar to the built-in type declarations.
- Structs are declared using the **struct** keyword.
- The simple form of a struct definition is as follows:

- struct *struct-name*
- {
- data member1;
- data member2;
- }
- *For Example:*
- struct Student {
- public string Name;
- public int RollNumber;
- public double TotalMarks;
- }
- The keyword **struct** declares **Student** as a new data type that can bold three variables of Types.

- These variables are known as *members or fields* or *elements.* The identifier **Student** can now be used to create variables of type Student. For *Example:*
- Student s1 ; //declare a student
- **s1** is a variable of type **Student** and has three member variables as defined by the template.

Assigning Values to Members

- Member variables can be accessed using the simple dot notation as follows:
- s1.Name ="John";
- s1.RollNumber = 999;
- s1.TotalMarks = 575.50;

Copying Structs

- We can also copy values from one struct *to* another. *For example:*
- Student s2; //s2 is declared
- s2 = s1 ;
- This will copy all those values from s1 *to* s2.
- We can also use the operator **new** to create **struct** variables.
- Student s3 = new Student ( );
- A **struct** variable is initialized to the default values of its members as soon as it is declared.
- Note that struct's data members are 'private' by default and there fore cannot be accessed ou[illegible]ct definition.

- **STRUCTS WITH METHODS**
- We have seen that values may be assigned to the data members using **struct** objects and the dot operator. We can also assign values to the data members using what are known as *constructors.*
- A constructor is a method which is used to set values of data members at the time of declaration.
- Consider the code below:
- sruct Number
- {
- int number; // data member
- public Number (int value) // constructor
- {
- number = value;
- }
- }

- The constructor method has the same name as struct and declared as public. The constructor is invoked as follows:
- Number n1 = new Number(100);
- This statement creates a struct object n1and assigns the value I00 to its only data member number.
- Structs can also have other methods as members. These methods may be designed to perform certain operations on the data stored in struct objects. Note that a struct is not permitted to declare a destructor.
- The following program shows how constructors and methods are used in a struct implementation.

- using System;
- struct Rectangle
- {

```
int a,b;
public Rectangle (int x, int y) //Constructor
{
   a = x;
   b = y;
   }
   public void Area( )   //a method
   {
      return(a * b);
   }
   public void Display ( )  //another method
   {
        Console.WritLine("Area ="+Area( ) );
   }
```

- Class TestRectangle
- {
-     public static void Main( )
-     {
-         Rectangle rect = new Rectangle (10,20);
-         rect.Display( ); // invoking Display
-     }
- }
- Program is produces the following output:
- Area= 200
- This code contains one constructor method to give values to the data members, another method **Area** ( ) to compute the area of the rectangle and the third method **Display** ( ) to ... computed.

- **NESTED STRUCTS**
- C# permits declaration of structs nested inside other structs. The following code is valid:

```
struct Employee
{
    public string name;
    public int code;
    public struct Salary
    {
        public double basic;
        public double allowance;
    }
}
```

- The following program is an example of nesting of strutctures in C#, one structure teacher is declared within structure student and their values are displayed using functions show ( ), show details( ), etc. The values of structure and method overriding is done using getvalues() method.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace NestedStructures
{
    struct student
    {
        public string studentname;
        public string rollno;
        public static string grade;
        //Setting the property with struct student
        public string RollNo
        {
```

- set
- {
- rollno = "5000018_ClassV"
- }
- get
- {
- return rollno;
- }
- }
- /* Declaring a class school with in the structure student public class school
- {
- public string schoolname;
- public string classname;
- …
- …

ENUMERATIONS

- An enumeration is a user-defined integer type which provides a way for attaching names to numbers, thereby increasing the comprehensibility of the code.
- The enum keyword automatically enumerates a list of words by assigning them values 0,1, 2 and so on.
- The syntax of an **enum** statement is illustrated below:
- enum Shape
- {
- Circle, //ends with comma
- Square, //ends with comma
- Triangle // no comma
- }
- This can be written in one line as follows:
- enum Shape{Circle, Square, Triangle}

- Here, **Circle** has the value 0, **Square** has the value 1 and **Triangle** has the value 2. Other examples of **enum** are:
- enum Colour { Red, Blue, Green, Yellow }
- enum Position { Off, On }
- enum Day { Mon, Tue, Wed, Thu, Fri, Sat, Sun}
- **ENUMERATOR INITIALIZAT ION**
- As mentioned earlier, bydefault, the value of the first enum member is set to 0, and that of each subsequent member is incremented by one. However, we may assign specific values for different members, if we so desire.

- *For Example:*
- enum Colour
- {
-     Red   = 1,
-     Blue =3,
-     Green = 7,
-     Yellow = 5
- }
- If the declaration of an enum member has no initializer, then its value is set implicitly as follows:
  - If it is the first member, its value is zero.
  - Otherwise, its value is obtained by adding one to the value of the previous member
- [illegible] following enum declaration:

- enum Alphabet
- {
- **A,**
- B=5
- C,
- D = 20,
- E
- }
- The member A is set to zero. Since the member B is explicitly given the value *5,* the value of C is set to 6 (i.e, *5* + 1). Similarly, E is set to 21.

- **ENUMERATOR BASETYPES**
- By default, the type of an enum is **int.** However , we can declare explicitly a base type for each enum . The valid base types are:
- byte, sbyte , short, ushort, int, uint, long and ulong
- *Examples:*
- enum Position : byte
- {
- off,
- on
- }
- The values assigned to the members must be within the range of values that can be represented by the base type. For example, if the base type is **byte,** assigning a value 300 is illegal.

- **ENUMERATOR TYPE CONVERSION**
- Enum types can be converted to their base type and back again with an explicit conversion using a cast.
- *Example:*
- enum Values
- {
- Value0,
- Value1 ,
- Value2,
- Value3
- }
- … …
- Values u1 = (Values) 1;
- int a = (int) u1;

- The exception to this is that the literal 0 can be converted to an enum type without a cast. That is,
- Values u0 = 0;
- is permitted. The following program illustrates how enumerator types are converted.
- using System;
- class Enumtype
- {
- enum Direction
- {

- North,
- East = 10;
- West,
- South
- }
- public static void Main ( )
- {
- Direction d1 = 0; //implicit conversion
- Direction d2 = Direction.East;
- Direction d3 = Direction.West;
- Direction d4 = (Direction) 12; //explicit
- Console.WriteLine(“d1=“+d1);

- Console.WriteLine(“d2=“+(int) d2)
- Console.WriteLine(“d3=“+d3);
- Console.WriteLine(“d4=“+d4);
- }
- }
- Program output will be as :
- d1 = 0
- d2 = 10
- d3 = 11
- d4 = 12