- Create an instance of the NullReferenceException class and then throw the ex object:

var ex = new NullReferenceException("The 'Name' parameter is null.");

throw ex;

-rethrow an exception that has been caught in a catch block and propagate it to the caller:

try{}

catch (NullReferenceException ex)

{ // Catch all NullReferenceException exceptions.}

catch (Exception ex){

// Attempt to handle the exception

...

// If this catch handler cannot resolve the exception,

// throw it to the calling code

throw;}

-----------------------------------------------------------------------------------------------------------------------------

**Arrays:**

- To declare a one-dimensional Array:

int[] arrayName = new int[arraySize]; or int[] arrayName= {1,2,...,10};

-multi-dimensional array:

int[ , ] arrayName = new int[10,10];

int[,] arrayName = {{1,2},{3,3}};

- element can be accessed: int value = arrayName[0,1];

**Jagged array:** is array of arrays with each has variable size.

int[][] jaggedArray = new int[10][];

jaggedArray[0] = new Type[5]; // array 0 has a size of 5

jaggedArray[1] = new Type[3]; // array 1 has a size of 3

**enum:** is a structure that enables you to create a variable with a fixed set of possible values. It has to be declared outside any method.

enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

By default enum values start at 0 and each successive member is increased by a value of 1.

like: Sunday =0, Monday=1,... .

You can change the default by specifying a starting value for your enum as in the following example.

enum Day { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }; the next elements will have increasing value.

In order to change the default data type of your enum, you precede the list with a data type to be either byte,sbyte,short,ushort,int,uint,long,ulong:

enum Day : short { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

----------------------------------------------------------------------------------------------------------------------------------------

**Structures:** structs are intended to be lightweight therefore if you find yourself adding multiple methods, constructors, and events, you should consider using a class instead.

//Declaring a Struct

public struct Coffee

{

public int Strength;

public string Bean;

public string CountryOfOrigin;

// Other methods, fields, properties, and events.

}

//instantiating a struct

Coffee coffee1 = new Coffee();

When you instantiate a struct, you are actually calling a special type of method called a constructor. A constructor is a method in the struct that has the same name as the struct.

When you instantiate a struct with no arguments, such as new Coffee(), you are calling the default constructor which is created by the Visual C# compiler. If you want to be able to specify default field values when you instantiate a struct, you can add constructors that accept parameters to your struct.

You cannot add a default constructor to a struct because it is created by the compiler.

public struct Coffee

{

// This is the custom constructor.

public Coffee(int strength, string bean, string countryOfOrigin)

{

this.Strength = strength;

this.Bean = bean;

this.CountryOfOrigin = countryOfOrigin;

}

// These statements declare the struct fields and set the default values.

public int Strength;

public string Bean;

public string CountryOfOrigin;

// Other methods, fields, properties, and events.

}

// Call the custom constructor

Coffee coffee1 = new Coffee(4, "Arabica", "Colombia");

coffee1.Strength = 3;

**Declaring struct:**

public struct Coffee

{

private int strength;

public int Strength

{

get { return strength; } // acessor method to read access to a field

set { strength = value; } // acessor to write access to a field

}

}

//Using a Property

Coffee coffee1 = new Coffee();

coffee1.Strength = 3; // The following code invokes the set accessor.

int coffeeStrength = coffee1.Strength; // The following code invokes the get accessor.

You can change the implementation of properties without affecting client code. For example, you can add validation logic, or call a method instead of reading a field value.

public int Strength

{

get { return strength; }

set

{

if(value < 1)

{ strength = 1; }

else if(value > 5)

{ strength = 5; }

else { strength = value; }

}

}

**Indexers:** it uses get and set accessors to control access to a field. It enables you to access members directly from the name of the containing struct or class by providing an integer.

//Creating an Indexer

public struct Menu

{

private string[] beverages;

// This is the indexer.

public string this[int index]

{

get { return this.beverages[index]; }

set { this.beverages[index] = value; }

}

// Enable client code to determine the size of the collection.

public int Length

{

get { return beverages.Length; }

}

}

To retrieve the value:

Menu myMenu = new Menu();

string firstDrink = myMenu[0]; // instead of =myMenu.beverges[0] without indexers

int numberOfChoices = myMenu.Length;

----------------------------------------------------------------------------------------------------------------------------------

**OOP - Classes:** Class is central to OOP. It enables encapsulate the behaviors and characteristics of any logical entity in a reusable manner.

The behavior and characteristics of the class is created by methods (events or functions) and fields (members). Class is like a blueprint for the type.

Properties store data for an object (get and set). Methods are actions an object can perform.

public class DrinksMachine

{

// The following statements define a property with a private field (using encapsulation. see next)

private string age;// right click,refractor, encapsulate field

public string Age

{

get{return age;}

set{age= value;}

}

public DrinksMachine(){Age =0;} // this is a constructor. It’s called whenever someone instantiates your class without providing any arguments.

// The following statements define properties.

public string Make

{get{return make;}

set

{ make = value;}

}; // to use: objectName.Make = some\_value

public string Model {};

// The following statements define methods.

public void MakeCappuccino()

{

// Method logic goes here.

}

public void MakeEspresso()

{

// Method logic goes here.

}

// The following statement defines an event. The delegate definition is not shown.

public event OutOfBeansHandler OutOfBeans;

}

// Instantiating a Class

DrinksMachine dm = new DrinksMachine();

// using dot notation to call members of the class

dm.Make = "Fourth Coffee";

dm.MakeEspresso();

When you instantiate a class this way:

-You are creating a new object in memory based on the DrinksMachine type.

-You are creating an object reference named dm that refers to the new DrinksMachine object.

When you create your object reference, instead of explicitly specifying the DrinksMachine type, you can allow the compiler to deduce the type of the object at compile time. This is known as type inference. To use type inference, you create your object reference by using the var keyword:

// Instantiating a Class by Using Type Inference

var dm = new DrinksMachine();

**Partial Classes:** enabling multi-teams to work on separate partial classes with some methods that belong to one class. ex:

public partial class DrinksMachine

{

public void MakeCappuccino(){// Method logic goes here.}

}

public partial class DrinksMachine

{

public void MakeEspresso(){// Method logic goes here.}

}

**Encapsulation:** can be used to describe the accessibility of the members belonging to a class or struct.

**private:**The type is only available to code within the class that contains it. You can only use the private access modifier with nested classes. This is the default value if you do not specify an access modifier.

**protected:**The type is only accessible within its class and by derived class instances.

The tradition is to create private data fields in the class to prevent direct manipulation of the values for those fields, and expose properties (accessors) to provide access to the values indirectly.

Properties enable you to permit users of the class a means of getting and setting values for the private member data fields within your class.

// auto-implemented property

public string Model { get; set; }

// Constructors

public DrinksMachine(int age)

{this.Age = age;}

public DrinksMachine(string make, string model)

{this.Make = make;

this.Model = model;}

-A get property accessor is used to return the property value

-A set accessor is used to assign a new value. (Omitting this property makes it read only)

-A value keyword is used to define the "value" being assigned by the set accessor.

-For simple properties that require no custom accessor code, consider the option of using auto-implemented properties.

A static class: is a class that cannot be instantiated. Sometimes you want to create a class purely to encapsulate some useful functionality, rather than to represent an instance of anything. Any members within the class must also use the static keyword. Can access a static method by calling the method.

#region // to collapse and show region

public static class Conversions

{

public static double PoundsToKilos(double pounds)

{// Convert argument from pounds to kilograms

double kilos = pounds \* 0.4536;

return kilos;}

}

#endregion

//Calling Methods on a Static Class using the class name without creating an instance

double weightInPounds = Conversions.KilosToPounds(80);

Some behaviors and characteristics relate to the instance (instance members), while some behaviors and characteristics relate to the type itself (static members).Methods, fields, properties, and events can all be declared static in a non-static class.

Static properties are often used to return data that is common to all instances, or to keep track of how many instances of a class have been created. Static methods are often used to provide utilities that relate to the type in some way, such as comparison functions.

**Anonymous classes:** offer the programmer a convenient way of encapsulating read-only properties into a single object without the need to explicitly define a type first.The type name will be generated by the compiler. The type name is also not available at the source code level and the type of each property included in this anonymous class will be inferred by the compiler.

To create anonymous class, use new keyword followed by a pair of braces to define fields and values for the class:

var anAnonymousObject = new { Name = "Tom", Age = 65 }; // The class will have two public fields, Name (initialized to the string “Tom”) and Age (initialized to 65). The compiler has inferred the types of these two fields based on the types of data you initialize them with.

Once instantiated, you can access the fields in the object by using dot notation, as shown in this example:

Console.WriteLine("Name: {0} Age: {1}", anAnonymousObject.Name, anAnonymousObject.Age};

The C# compiler will look at the names, types, number, and the order of the fields in the object in order to determine whether two instances of an anonymous class have the same type or not. In our two examples, both objects contain the same number of fields, the same name and the same type, in the same order. As a result, both variables are instances of the same anonymous class. This means that you can assign anAnonymousObject to the secondAnonymousObject or vice versa:

secondAnonymousObject = anAnonymousObject;

**However,**

-anonymous classes can contain only public fields

-the fields must all be initialized

-fields cannot be static

-you cannot define any methods for them