CLASS: -

A **class** is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A **class** is like a blueprint. It defines the data and behavior of a type.

OBJECT:-

A class or struct definition is like a blueprint that specifies what the type can do. An **object** is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many **objects** of the same class.

DIFFERENT TYPES OF CLASSES IN ASP.NET :-

• Abstract Class (somtimes called a Pure Virtual Class)  
  
• Partial Class  
  
• Sealed Class  
  
• Static Class

1. **Abstract Class:-**

An Abstract Class means that, no object of this class can be instantiated, but can make derivation of this. It can serve the purpose of base class only as no object of this class can be created.Abstract Class is denoted by the keyword *abstract*.

EX.

abstract class myClass  
{  
 public myClass()  
{  
 // code to initialize the class…  
}  
  
 abstract public void anyMethod\_01();  
 abstract public void anyMethod\_02(int anyVariable);  
 abstract public int anyMethod\_03 (int anyvariable);  
}

It is important here to note that abstract classes can have non-abstract method(s), even can have only non-abstract method(s).

abstract class myclass  
{  
 public void nonAbstractMethod ()  
 {  
 // code…  
 }  
}

… is perfectly alright.  
An abstract class can be derived from another abstract class. In that case, in the derived class, it is optional to implement the abstract method(s) of the base class.

### Example:

// Base abstract class  
abstract class baseClass  
{  
 public abstract int addNumbers (int a, int b);  
 public abstract int multiplyNumbers(int a, int b);  
}  
  
// Derived abstract class  
abstract class derivedClass:baseClass  
{  
 // implementing addNumbers…  
 public override int addnumbers (int a, int b)  
 {  
 return a+b;  
 }  
}  
  
// Derived class from 2nd class (derivedClass)  
class anyClass: derivedClass  
{  
 // implementing multiplyNumbers of baseClass…  
 public override int multiplyNumbers(int a, int b)  
 {  
 return a\*b;  
 }  
}

In the above example, we only implemented *addNumbers* in the derived abstract class (*derivedClass*). The abstract method *multiplyNumbers* is implemented in the *anyClass*, which is in turn derived from *derivedClass*.

## Partial Class:

This special type of class called "Partial Class" is introduced with .Net Framework 2.0. Partial Class allows its members – method, properties, and events – to be divided into multiple source files (.cs). At compile time these files get combined into a single class.  
Partial Class is denoted by the keyword *partial*.  
Some do's and don'ts about partial class:-  
• All the parts of a partial class must be prefixed with the keyword partial.  
• Accessibility, signature etc. must be same in all parts of the partial class.  
• You cannot sealed one part of the partial class. In that case entire class in sealed.  
• If you define any part of the partial class abstract, entire class will become abstract.   
• Inheritance cannot be applied to a part of partial class. If you do so, it applies to entire class.

### Example:

public partial class myPartialClass  
{  
 public void firstMethod()  
 {  
 // code…  
 }  
}  
  
public partial class myPartialClass  
{  
 public void secondMethod()  
 {  
 // code…  
 }  
}

## Sealed Class:

A sealed class is a class which cannot be inherited. A sealed class cannot be a base class. The modifier abstract cannot be applied to a sealed class. By default, struct (structure) is sealed. It is the last class in hierarchy. To access the members of a sealed class, you must create objects of that class.   
Sealed Class is denoted by the keyword *sealed*.

### Example:

sealed class mySealedClass  
{  
 int a;  
 int b;  
}  
  
Class mainClass  
{  
 public static void Main()  
 {  
 mySealedClass obj = new mySealedClass();  
 obj.a = 5;  
 obj.b = 7;  
 Console.WriteLine("a = {0}, b = {1}", obj.a, obj.b);  
 }  
}

## Static Class:

A Static Class is one which cannot be instantiated. The keyword new cannot be used with static classes as members of such class can be called directly by using the class name itself.  
Following are the main characteristics of a static class:-  
• A Static Class can only have static members.  
• A Static Class cannot be instantiated.   
• A Static Class is sealed, so cannot be inherited.  
• A Static Class cannot have a constructor (except static constructor).  
Static Class is denoted by the keyword *static*.

### Example:

// static class definition…  
public static class myclass  
{  
 public static int addNumbers(int a, int b)  
 {  
 return (a + b);  
 }  
}  
  
// to use it, we call directly on the class…  
Console.WriteLine("The addition of 5 and 7 is: " + myClass.addNumbers(5, 7));

**VALUE TYPE:-**

Variables that store data are called value types. Value types are stored on stack.  
They contain the actual values. eg-int, enum, structs.

**OR**

A Value Type stores its contents in memory allocated on the stack. When you created a Value Type, a single space in memory is allocated to store the value and that variable directly holds a value. If you assign it to another variable, the value is copied directly and both variables work independently. Predefined datatypes, structures, enums are also value types, and work in the same way. Value types can be created at compile time and Stored in stack memory, because of this, Garbage collector can't access the stack.

e.g. int x = 10;

Here the value 10 is stored in an area of memory called the stack

**REFERENCE TYPE:-**

Variables that store reference to actual data are called Reference types.Reference types  
stored on heap but contain the address on heap.  
eg-class,interface,delegate,string,object**,** Array

OR :

Reference Types are used by a reference which holds a reference (address) to the object but not the object itself. Because reference types represent the address of the variable rather than the data itself, assigning a reference variable to another doesn't copy the data. Instead it creates a second copy of the reference, which refers to the same location of the heap as the original value. Reference Type variables are stored in a different area of memory called the heap. This means that when a reference type variable is no longer used, it can be marked for garbage collection. Examples of reference types are Classes, Objects, Arrays, Indexers, Interfaces etc.

e.g. int[] iArray = new int[20];

In the above code the space required for the 20 integers that make up the array is allocated on the heap.

EXAMPLE:-

class Program  
{  
    static void Main(string[] args)  
    {  
        // Pass reference type by value  
        ArrayList arrayList = new ArrayList() { 0, 1, 2, 3 };  
         Console.WriteLine("Pass by Value");  
   
        PassByValue(arrayList);

        // What should be the output of below line ??  
        Console.WriteLine(arrayList[1]);  
                arrayList = new ArrayList() { 0, 1, 2, 3 };

             Console.WriteLine("Pass by Reference");

        PassByReference(ref arrayList);

        // What should be the output of below line ??  
        Console.WriteLine(arrayList[1]);

        Console.Read();  
    }

    private static void PassByValue(ArrayList arrayList)  
    {

        Console.WriteLine(arrayList[1]);  
        // Now Change the first position value  
        arrayList[1] = 90;  
        arrayList = new ArrayList() { 101, 102, 103, 104 };

        Console.WriteLine(arrayList[1]);  
    }

    private static void PassByReference(ref ArrayList arrayList)  
    {

        Console.WriteLine(arrayList[1]);

        // Now Change the first position value  
        arrayList[1] = 90;  
             arrayList = new ArrayList() { 101, 102, 103, 104 };

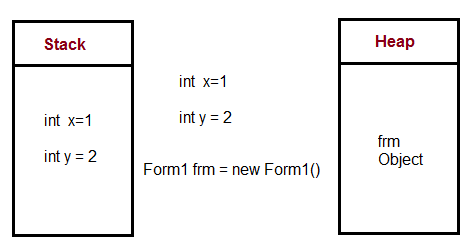
        Console.WriteLine(arrayList[1]);  
    }  
}

**Stack and a Heap ?**

Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM .

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled. When a function or a method calls another function which in turns calls another function etc., the execution of all those functions remains suspended until the very last function returns its value. The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory . Element of the heap have no dependencies with each other and can always be accessed randomly at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.



You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

In a multi-threaded situation each thread will have its own completely independent stack but they will share the heap. Stack is thread specific and Heap is application specific. The stack is important to consider in exception handling and thread executions.

OR: -

|  |  |
| --- | --- |
| **Stack** | **Heap** |
| Values are stored on one another like a stack. | Values are stored in random order.like dumped into a huge space |
| Used for value type | Used for reference types |

**CONSTRUCTOR :**

Constructor is a method that is called when an instance of an object is created.   
 They have the same name as a class.  
    eg-  
        class Test  
       {  
             public Test()  
            {  
            }  
        }

**Can a constructor have access modifier ?**  
    Yes .The foll are access modifiers allowed -  
     1.Public --This is called whenever a class is initialized.  
     2.Private- This will prevent the class from being instantiated(creating objects)  
 3.Protected-  
     4.Internal--cannot be instantiated outside the assembly.