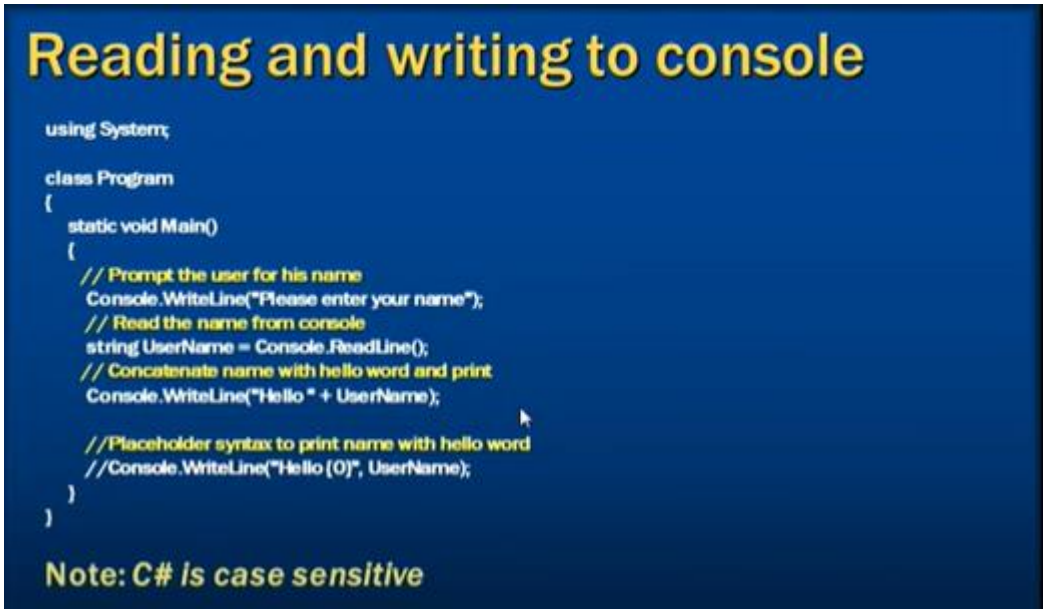
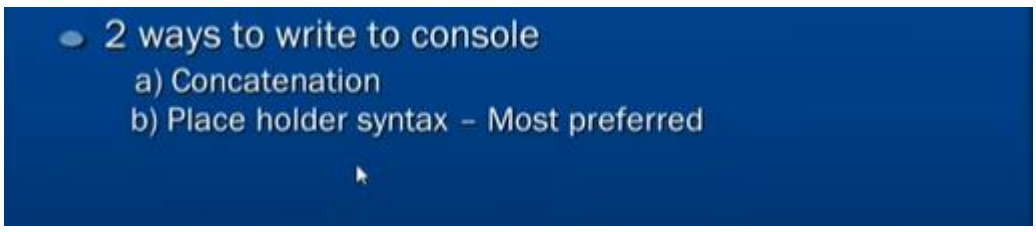
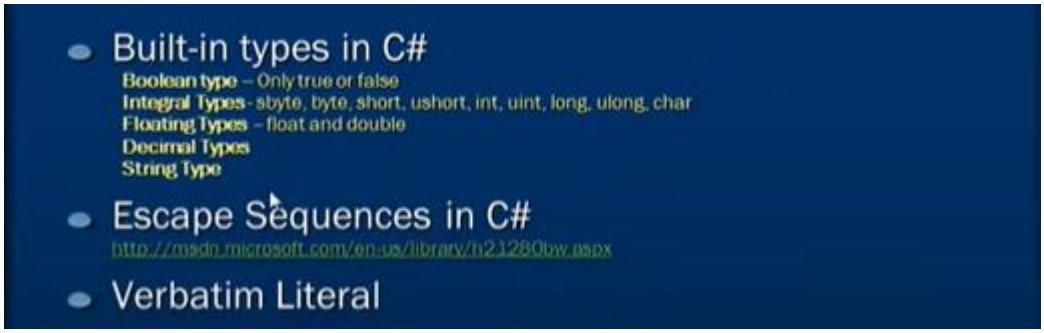
Namespace is collection of related Classes, Enums, Delegates, Namespace, Structs etc.

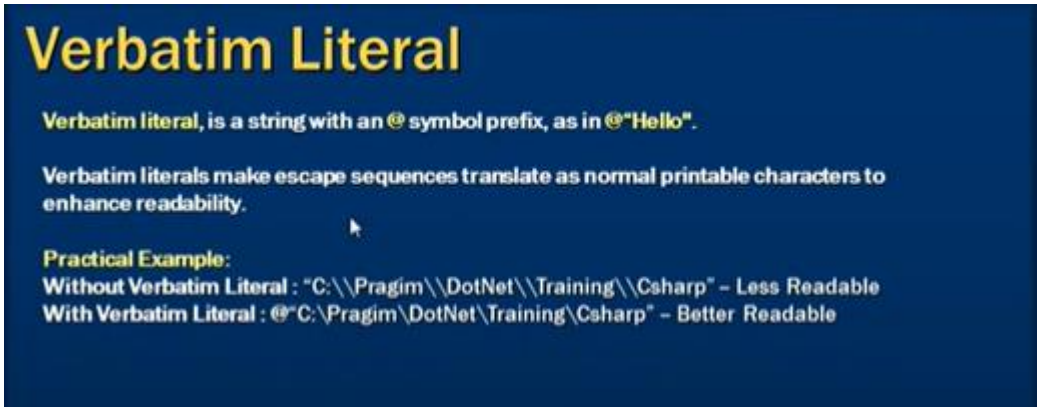


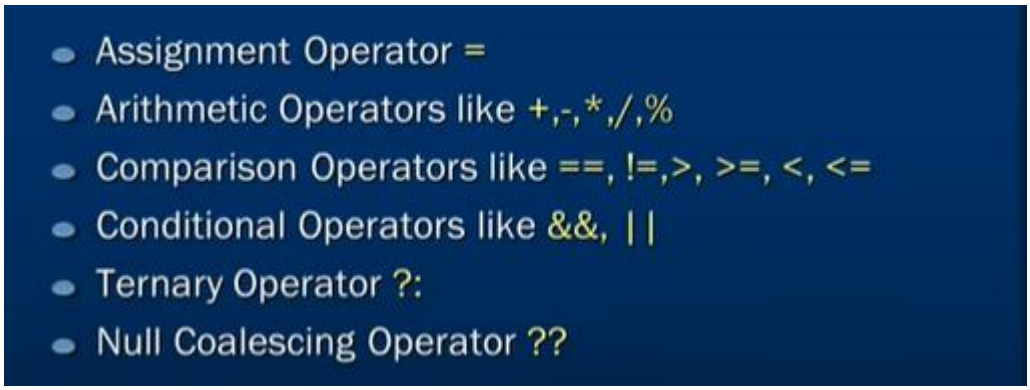


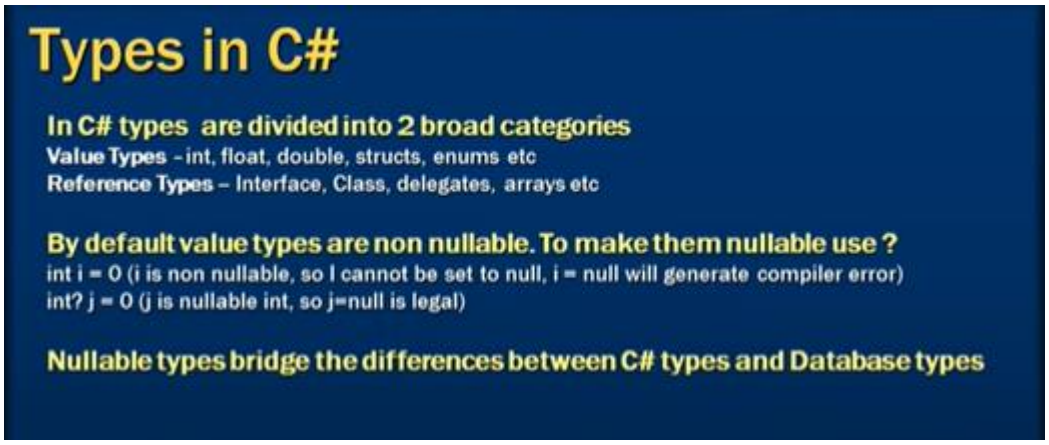


C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
`nint`	Depends on platform	Signed 32-bit or 64-bit integer	System.IntPtr
`nuint`	Depends on platform	Unsigned 32-bit or 64-bit integer	System.UIntPtr

C# type/keyword	Approximate range	Precision	Size	.NET type
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	System.Single
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	System.Double
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	System.Decimal







= value ?? default value (If value is null then default value will get assigned to left side variable)

Implicit & Explicit conversion

Implicit conversion is done by the compiler:

1. When there is no loss of information if the conversion is done
2. If there is no possibility of throwing exceptions during the conversion

Example: Converting an int to a float will not loose any data and no exception will be thrown, hence an implicit conversion can be done.

Where as when converting a float to an int, we loose the fractional part and also a possibility of overflow exception. Hence, in this case an explicit conversion is required. For explicit conversion we can use cast operator or the convert class in c#

Explicit Conversion Example

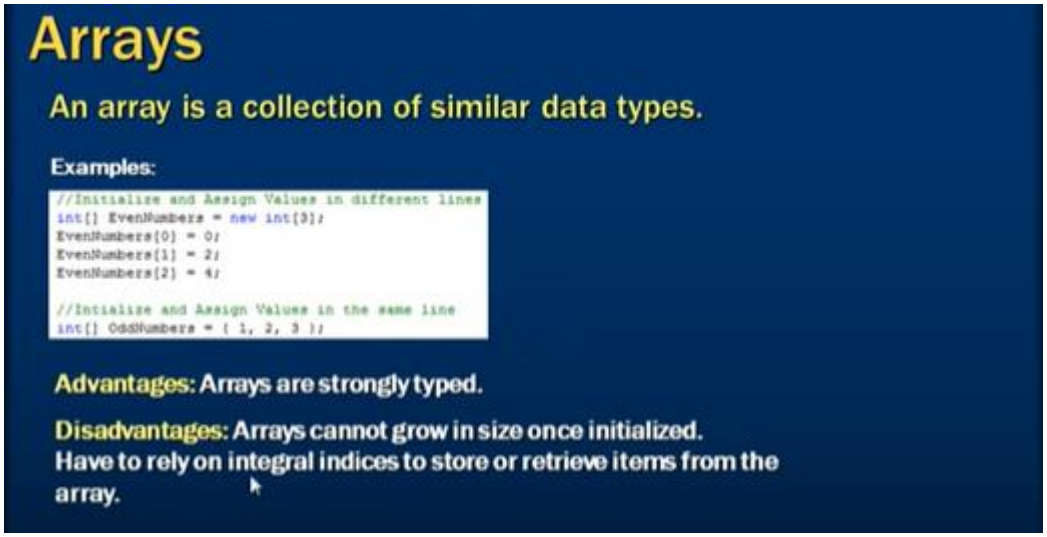
```
using System;
class Program
{
    static void Main()
    {
        float f = 100.25F;

        // Cannot implicitly convert float to int
        // Fractional part will be lost. Float is a
        // bigger data type than int, so there is
        // also a possibility of overflow exception
        // int i = f;

        // Use explicit Conversion using cast () operator
        int i = (int)f;

        // Or use Convet class
        // int i = Convert.ToInt32(f);

        Console.WriteLine(i);
    }
}
```

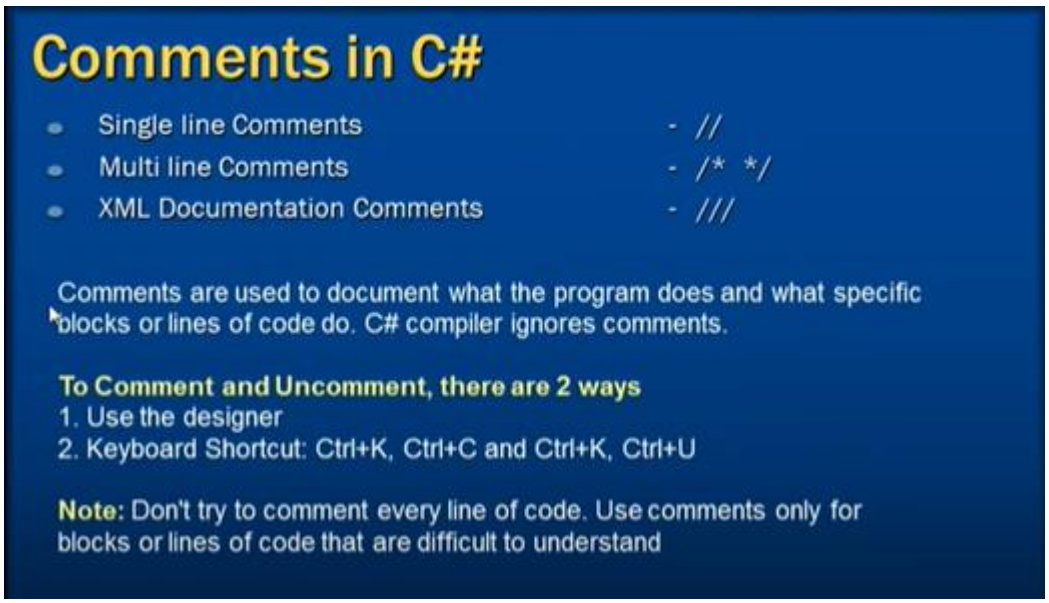
Difference between Parse and TryParse

If the number is in a string format you have 2 options - Parse() and TryParse()

Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicating whether it succeeded or failed.

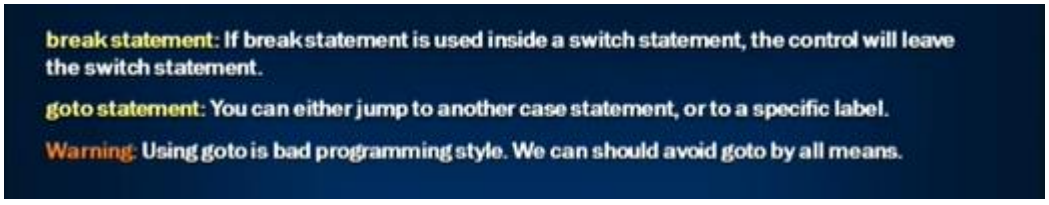
Use Parse() if you are sure the value will be valid, otherwise use TryParse()





Short Circuit Expressions : && ||

&,| ➔ executes both the expression while &&,|| skips others based on the condition



```
Start:
Console.WriteLine("Please Select your coffee size : 1 - Small, 2 - Medium, 3 - Large");
int UserChoice = int.Parse(Console.ReadLine());

switch (UserChoice)
{
    case 1:
        TotalCoffeeCost += 1;
        break;
    case 2:
        TotalCoffeeCost += 2;

        goto case 1;
    case 3:
        TotalCoffeeCost += 3;
        break;
    default:
        Console.WriteLine("Your choice {0} is invalid", UserChoice);
        goto Start;
}
```

- while
- do, for, foreach
- Difference between while and do loop

The while loop

1. While loop checks the condition first.
2. If the condition is true, statements with in the loop are executed.
3. This process is repeated as long as the condition evaluates to true.

Note: Don't forget to update the variable participating in the condition, so the loop can end, at some point

Do While loop

1. A do loop checks its condition at the end of the loop.
2. This means that the do loop is guaranteed to execute at least one time.
3. Do loops are used to present a menu to the user

Difference – while & do while

1. While loop checks the condition at the beginning, where as do while loop checks the condition at the end of the loop.
2. Do loop is guaranteed to execute at least once, where as while loop is not.

The for loop

A for loop is very similar to while loop. In a while loop we do the initialization at one place, condition check at another place, and modifying the variable at another place, where as for loop has all of these at one place.

foreach loop

A foreach loop is used to iterate through the items in a collection. For example, foreach loop can be used with arrays or collections such as ArrayList, HashTable and Generics. We will cover collections and generics in a later session.

Why methods

Methods are also called as functions. These terms are used interchangeably.

Methods are extremely useful because they allow you to define your logic once, and use it, at many places.

Methods make the maintenance of your application easier.

Methods

```
[attributes]
access-modifiers return-type method-name ( parameters )
{
   Method Body
}
```

1. We will talk about attributes and access modifiers in a later session
2. Return type can be any valid data type or void.
3. Method name can be any meaningful name
4. Parameters are optional

Static Vs Instance methods

When a method declaration includes a static modifier, that method is said to be a static method. When no static modifier is present, the method is said to be an instance method.

Static method is invoked using the class name, where as an instance method is invoked using an instance of the class.

The difference between instance methods and static methods is that multiple instances of a class can be created (or instantiated) and each instance has its own separate method. However, when a method is static, there are no instances of that method, and you can invoke only that one definition of the static method.

Method parameter types

There are 4 different types of parameters a method can have

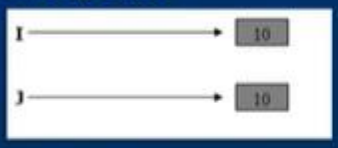
1. **Value Parameters** : Creates a copy of the parameter passed, so modifications does not affect each other.
2. **Reference Parameters** : The ref method parameter keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.
3. **Out Parameters** : Use when you want a method to return more than one value.
4. **Parameter Arrays** :The params keyword lets you specify a method parameter that takes a variable number of arguments. You can send a comma-separated list of arguments, or an array, or no arguments. Params keyword should be the last one in a method declaration, and only one params keyword is permitted in a method declaration.

Note : Method Parameter Vs Method Argument

```
class Program
{
    public static void Main()
    {
        int i = 0;

        SimpleMethod(ref i);

        Console.WriteLine(i);
    }

    public static void SimpleMethod(ref int j)
    {
        j = 101;
    }
}
```

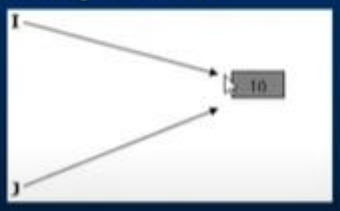
Pass by value/reference

Pass by value :



I and J are pointing to different memory locations. Operations one variable will not affect the value of the other variable.

Pass by reference :



I and J are pointing to the same memory location. Operations one variable will affect the value of the other variable.

```
using System;

class Program
{
    public static void Main()
    {
        int Total = 0;
        int Product = 0;
        Calculate(10, 20, out Total, out Product);

        Console.WriteLine("Sum = {0} && Product = {1}", Total, Product);
    }

    public static void Calculate(int FN, int SN, out int Sum, out int Product)
    {
        Sum = FN + SN;
        Product = FN * SN;
    }
}
```

```
using System;

class Program
{
    public static void Main()
    {
        int[] Numbers = new int[3];

        Numbers[0] = 101;
        Numbers[1] = 102;
        Numbers[2] = 103;

        //ParamsMethod();
        //ParamsMethod(Numbers);
        ParamsMethod(1, 2, 3, 4, 5);
    }

    public static void ParamsMethod(params int[] Numbers)
    {
        Console.WriteLine("There are {0} elemnts", Numbers.Length);

        foreach (int i in Numbers)
        {
            Console.WriteLine(i);
        }
    }
```

Why Namespaces

Namespaces are used to organize your programs.

They also provide assistance in avoiding name clashes.

Namespaces

Namespaces don't correspond to file, directory or assembly names. They could be written in separate files, and/or separate assemblies and still belong to the same namespace.

Namespaces can be nested in 2 ways.

Namespace alias directives. Sometimes you may encounter a long namespace and wish to have it shorter. This could improve readability and still avoid name clashes with similarly named methods.

Both are same

```
namespace ProjectA.TeamA
{

}
```

```
namespace ProjectA
{
    namespace TeamA
    {

    }
}
```

What is a class?

So far in this video tutorial we have seen simple data types like int, float, double etc. If you want to create complex custom types, then we can make use of classes.

A class consists of data and behavior. Class data is represented by it's fields and behavior is represented by its methods.

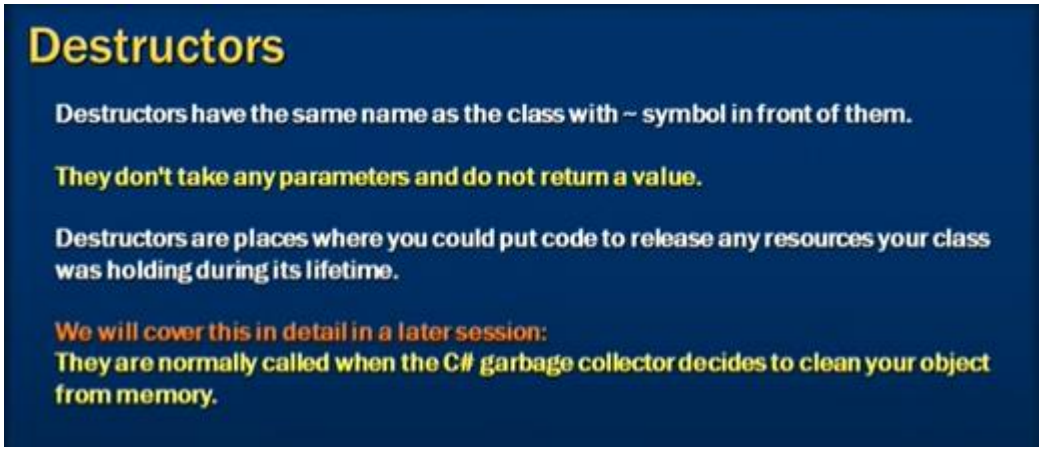
Purpose of a class constructor

The purpose of a class constructor is to initialize class fields. A class constructor is automatically called when an instance of a class is created.

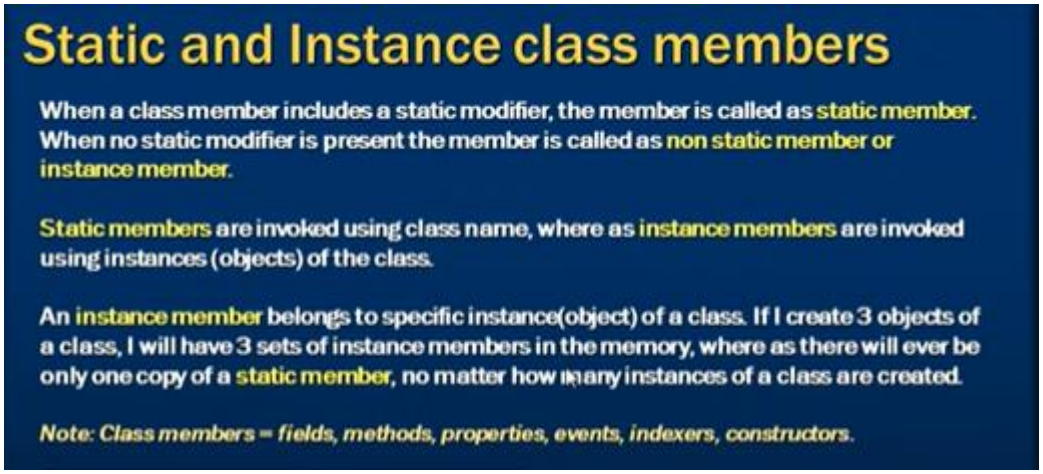
Constructors do not have return values and always have the same name as the class.

Constructors are not mandatory. If we do not provide a constructor ,a default parameter less constructor is automatically provided.

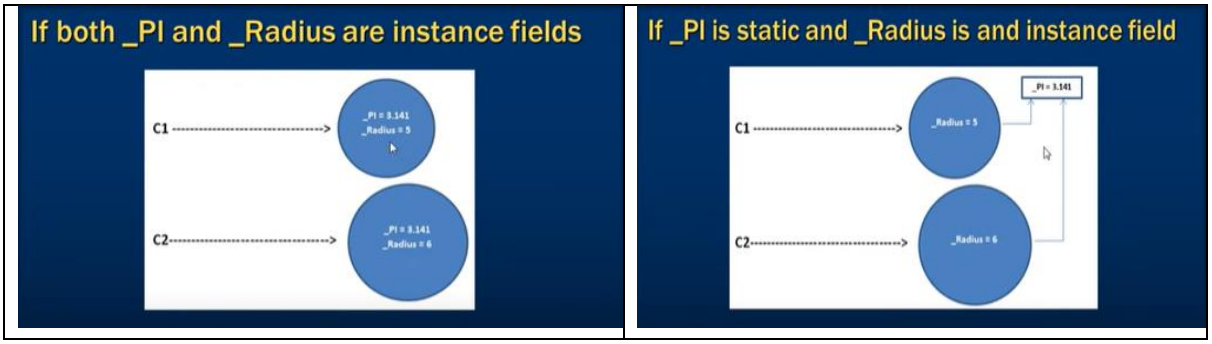
Constructors can be overloaded by the number and type of parameters.




```
class Customer
{
    string _firstName;
    string _lastName;

    public Customer()
        : this("No FirstName Provided", "No LastName Provided")
    {
    }

    public Customer(string FirstName, string LastName)
    {
        this._firstName = FirstName;
        this._lastName = LastName;
    }

    public void PrintFullName()
    {
        Console.WriteLine("Full Name = {0}", this._firstName + " " + this._lastName);
    }

    ~Customer()
    {
```

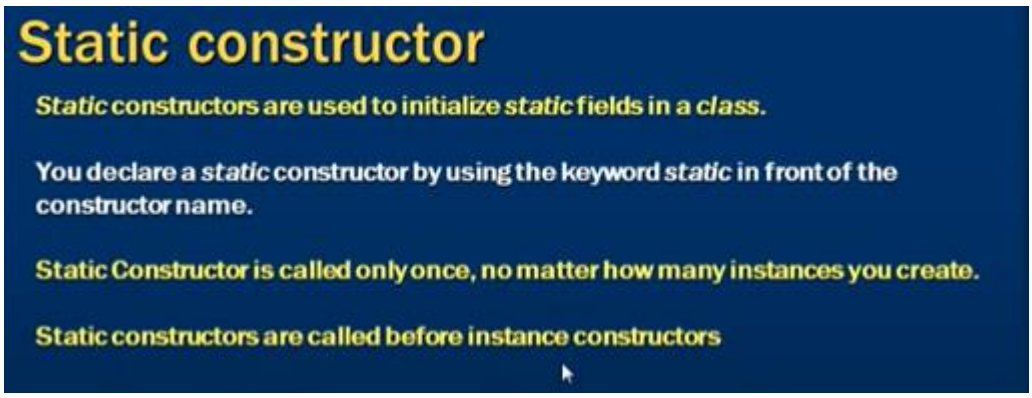


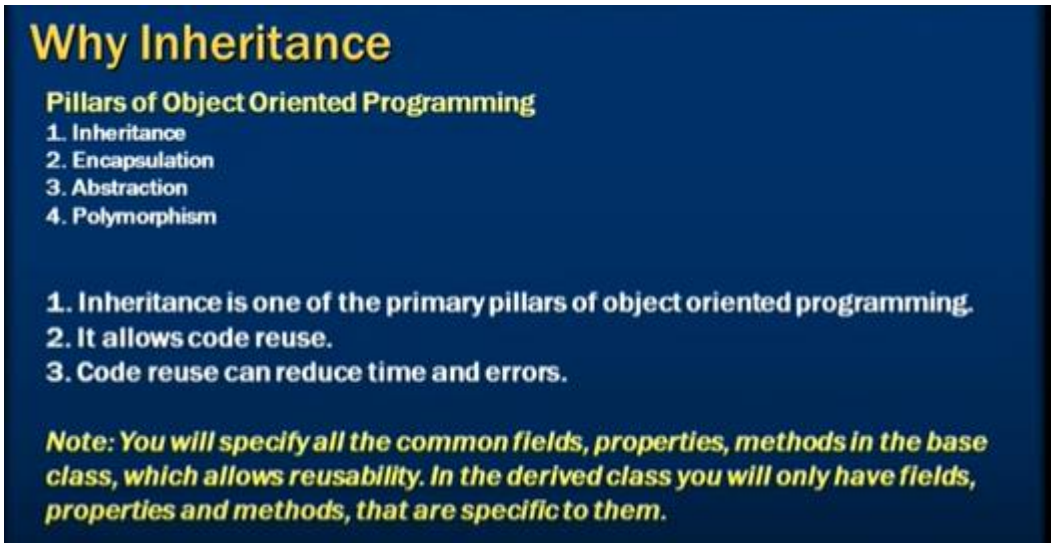
Static property: Property which is not going to change with respect to the instances so its better to create it as a static one, another benefit of doing so is that it will avoid creating multiple memory block for same property in each instance in heap

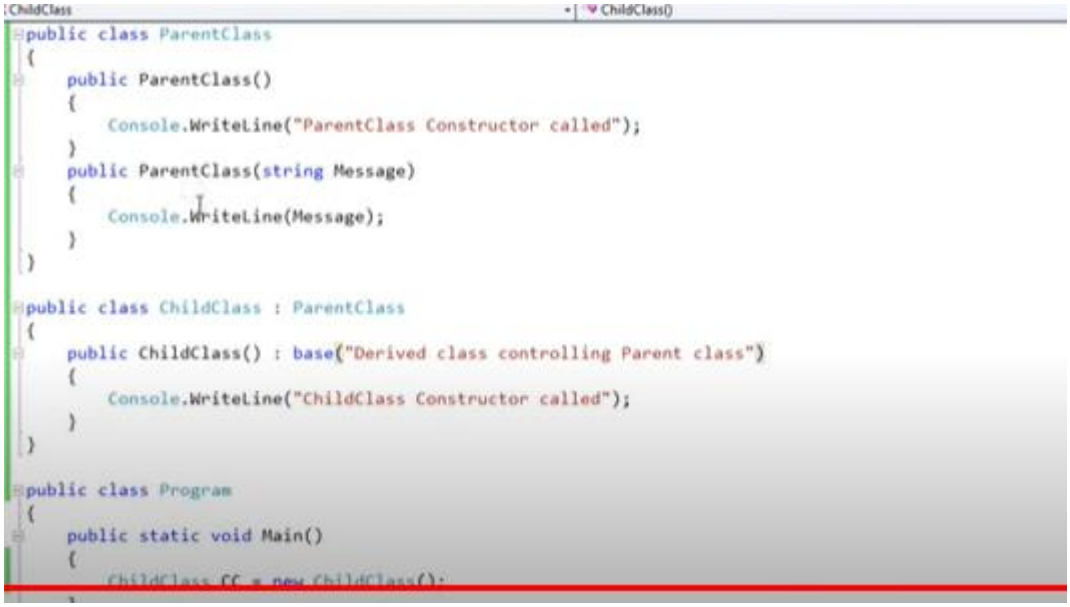


To initialize the instance member, we can initialize it in Instance Constructor, similarly, to initialize the static member we can initialize it in static constructor.

Static constructor cannot have any Access Modifier. And called automatically before accessing any of the fields of the Class, Called only once.







```
ChildClass                                          ChildClass()
public class ParentClass
{
    public ParentClass()
    {
        Console.WriteLine("ParentClass Constructor called");
    }
    public ParentClass(string Message)
    {
        Console.WriteLine(Message);
    }
}

public class ChildClass : ParentClass
{
    public ChildClass() : base("Derived class controlling Parent class")
    {
        Console.WriteLine("ChildClass Constructor called");
    }
}

public class Program
{
    public static void Main()
    {
        ChildClass CC = new ChildClass();
```

Inheritance Concepts

Use the new keyword to hide a base class member. You will get a compiler warning, if you miss the new keyword.

Different ways to invoke a hidden base class member from derived class

1. Use base keyword
2. Cast child type to parent type and invoke the hidden member
3. ParentClass PC = new ChildClass()
   PC.HiddenMethod()

Polymorphism

Polymorphism is one of the primary pillars of object-oriented programming.

Polymorphism allows you to invoke derived class methods through a base class reference during runtime.

In the base class the method is declared virtual , and in the derived class we override the same method.

The virtual keyword indicates, the method can be overridden in any derived class.

Method overriding Vs Method Hiding

```
public class BaseClass
{
    public virtual void Print()
    {
        Console.WriteLine("Base Class Print Method");
    }
}
public class DerivedClass : BaseClass
{
    public override void Print()
    {
        Console.WriteLine("Child Class Print Method");
    }
}
public class Program
{
    public static void Main()
    {
        BaseClass B = new DerivedClass();
        B.Print();
    }
}
```

In method overriding a base class reference variable pointing to a child class object, will invoke the overridden method in the Child class

```
public class BaseClass
{
    public virtual void Print()
    {
        Console.WriteLine("Base Class Print Method");
    }
}
public class DerivedClass : BaseClass
{
    public new void Print()
    {
        Console.WriteLine("Child Class Print Method");
    }
}
public class Program
{
    public static void Main()
    {
        BaseClass B = new DerivedClass();
        B.Print();
    }
}
```

In method hiding a base class reference variable pointing to a child class object, will invoke the hidden method in the Base class

Method overloading

Function overloading and Method overloading terms are used interchangeably.

Method overloading allows a class to have multiple methods with the same name, but, with a different signature. So, in C# functions can be overloaded based on the number, type(int, float etc) and kind(Value, Ref or Out) of parameters.

The signature of a method consists of the name of the method and the type, kind (value, reference, or output) and the number of its formal parameters. The signature of a method does not include the return type and the params modifier. So, it is not possible to overload a function, just based on the return type or params modifier.

Why Properties

Marking the class fields public and exposing to the external world is bad, as you will not have control over what gets assigned and returned.

```
public class Student
{
    public int ID;
    public string Name;
    public int PassMark;
}
public class Program
{
    public static void Main()
    {
        Student C1 = new Student();
        C1.ID = -101;
        C1.Name = null;
        C1.PassMark = -100;
        Console.WriteLine("ID = {0} & Name = {1} & PassMark = {2}",
            C1.ID, C1.Name, C1.PassMark);

    }
}
```

Problems with Public Fields

1. ID should always be non negative number
2. Name cannot be set to NULL
3. If Student Name is missing "No Name" should be returned
4. PassMark should be read only

Properties

In C# to encapsulate and protect fields we use properties

1. We use get and set accessors to implement properties
2. A property with both get and set accessor is a Read/Write property
3. A property with only get accessor is a Read only property
4. A property with only set accessor is a Write only property

Note: The advantage of properties over traditional Get() and Set() methods is that, you can access them as if they were public fields

Auto Implemented Properties

If there is no additional logic in the property accessors, then we can make use of auto-implemented properties introduced in C# 3.0.

Auto-implemented properties reduce the amount of code that we have to write.

When you use auto-implemented properties, the compiler creates a private, anonymous field that can only be accessed through the property's get and set accessors.

Structs

Just like classes structs can have

1. Private Fields
2. Public Properties
3. Constructors
4. Methods

Object initializer syntax, introduced in C# 3.0 can be used to initialize either a struct or a class.

Classes Vs Structs

A struct is a value type where as a class is a reference type.

All the differences that are applicable to value types and reference types are also applicable to classes and structs.

Structs are stored on stack, where as classes are stored on the heap.

Value types hold their value in memory where they are declared, but reference types hold a reference to an object in memory.

Value types are destroyed immediately after the scope is lost, where as for reference types only the reference variable is destroyed after the scope is lost. The object is later destroyed by garbage collector. (We will talk about this in the garbage collection session)

When you copy a struct into another struct, a new copy of that struct gets created and modifications on one struct will not affect the values contained by the other struct.

When you copy a class into another class, we only get a copy of the reference variable. Both the reference variables point to the same object on the heap. So, operations on one variable will affect the values contained by the other reference variable.

Structs can't have destructors, but classes can have destructors.

Structs cannot have explicit parameter less constructor where as a class can.

Struct can't inherit from another class where as a class can, Both structs and classes can inherit from an interface.

Examples of structs in the .NET Framework - int (System.Int32), double(System.Double) etc.

Note 1: A class or a struct cannot inherit from another struct. Struct are sealed types.
Note 2: How do you prevent a class from being inherited? Or What is the significance of sealed keyword?

Structs Vs Class Implementation

```
public class Customer
{
    // Private Fields
    private int _id;
    private string _name;
    // Constructor
    public Customer(int Id, string Name)
    {
        this._id = Id;
        this._name = Name;
    }
    // Public Properties
    public int ID
    {
        get { return this._id;}
        set { this._id = value;}
    }
    public string Name
    {
        get{return this._name;}
        set{this._name = value;}
    }
    // Method
    public void PrintName()
    {
        Console.WriteLine("Id = {0} && Name = {1}",
            this._id, this._name);
    }
}
```

```
public struct Customer
{
    // Private Fields
    private int _id;
    private string _name;
    // Constructor
    public Customer(int Id, string Name)
    {
        this._id = Id;
        this._name = Name;
    }
    // Public Properties
    public int ID
    {
        get { return this._id;}
        set { this._id = value;}
    }
    public string Name
    {
        get{return this._name;}
        set{this._name = value;}
    }
    // Method
    public void PrintName()
    {
        Console.WriteLine("Id = {0} && Name = {1}",
            this._id, this._name);
    }
}
```

*Structs always have the default constructor

Interfaces

We create interfaces using interface keyword. Just like classes interfaces also contains properties, methods, delegates or events, but only declarations and no implementations.

It is a compile time error to provide implementations for any interface member.

Interface members are public by default, and they don't allow explicit access modifiers.

Interfaces cannot contain fields.

If a class or a struct inherits from an interface, it must provide implementation for all interface members. Otherwise, we get a compiler error.

A class or a struct can inherit from more than one interface at the same time, but where as, a class cannot inherit from more than once class at the same time.

Interfaces can inherit from other interfaces. A class that inherits this interface must provide implementation for all interface members in the entire interface inheritance chain.

We cannot create an instance of an interface, but an interface reference variable can point to a derived class object.

Interface Naming Convention: Interface names are prefixed with capital I.

Abstract classes

The abstract keyword is used to create abstract classes.

An abstract class is incomplete and hence cannot be instantiated.

An abstract class can only be used as base class.

An abstract class cannot be sealed.

An abstract class may contain abstract members(methods, properties, indexers, and events), but not mandatory.

A non-abstract class derived from an abstract class must provide implementations for all inherited abstract members.

If a class inherits an abstract class, there are 2 options available for that class
Option 1: Provide Implementation for all the abstract members inherited from the base abstract class.
Option 2: If the class does not wish to provide Implementation for all the abstract members inherited from the abstract class, then the class has to be marked as abstract.

Abstract classes Vs Interfaces

Abstract classes can have implementations for some of its members (Methods), but the interface can't have implementation for any of its members.

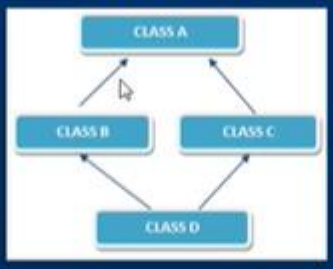
Interfaces cannot have fields where as an abstract class can have fields.

An interface can inherit from another interface only and cannot inherit from an abstract class, where as an abstract class can inherit from another abstract class or another interface.

A class can inherit from multiple interfaces at the same time, where as a class cannot inherit from multiple classes at the same time.

Abstract class members can have access modifiers where as interface members cannot have access modifiers.

Multiple Class Inheritance Problem



1. Class B and Class C inherit from Class A.
2. Class D inherits from both B and C.
3. If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?

This ambiguity is called as Diamond problem

Multiple Class Inheritance using interfaces

```
interface IA
{
    void AMethod();
}
class A : IA
{
    public void AMethod()
    {
        Console.WriteLine("A");
    }
}

interface IB
{
    void BMethod();
}
class B : IB
{
    public void BMethod()
    {
        Console.WriteLine("B");
    }
}
```

```
class AB : IA, IB
{
    A a = new A();
    B b = new B();
    public void AMethod()
    {
        a.AMethod();
    }
    public void BMethod()
    {
        b.BMethod();
    }
}
```

```
class Program
{
    public static void Main()
    {
        AB ab = new AB();
        ab.AMethod();
        ab.BMethod();
    }
}
```

A delegate is a type-safe function point

What is a delegate

A delegate is a type safe function pointer. That is, it holds a reference (Pointer) to a function.

The signature of the delegate must match the signature of the function, the delegate points to, otherwise you get a compiler error. This is the reason delegates are called as type safe function pointers.

A Delegate is similar to a class. You can create an instance of it, and when you do so, you pass in the function name as a parameter to the delegate constructor, and it is to this function the delegate will point to.

Tip to remember delegate syntax: Delegates syntax look very much similar to a method with a delegate keyword.

Delegate example

```
using System;

public delegate void HelloFunctionDelegate(string Message);

class Pragim
{
    public static void Main()
    {
        HelloFunctionDelegate del = new HelloFunctionDelegate(Hello);
        del("Hello from Delegte");
    }

    public static void Hello(string strMessge)
    {
        Console.WriteLine(strMessge);
    }
}
```

Multicast Delegates

A Multicast delegate is a delegate that has references to more than one function. When you invoke a multicast delegate, all the functions the delegate is pointing to, are invoked.

There are 2 approaches to create a multicast delegate. Depending on the approach you use

+ or += to register a method with the delegate

- or -= to un-register a method with the delegate

Note: A multicast delegate, invokes the methods in the invocation list, in the same order in which they are added.

If the delegate has a return type other than void and if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter, the value of the output parameter, will be the value assigned by the last method.

Common interview question - Where do you use multicast delegates?
Multicast delegate makes implementation of observer design pattern very simple. Observer pattern is also called as publish/subscribe pattern.

Multicast Delegate Examples

```
public delegate void SampleDelegate();

public class Sample
{
    static void Main()
    {
        SampleDelegate del1 = new SampleDelegate(SampleMethodOne);
        SampleDelegate del2 = new SampleDelegate(SampleMethodTwo);
        SampleDelegate del3 = new SampleDelegate(SampleMethodThree);

        SampleDelegate del4 = del1 + del2 + del3 - del2;

        del4();
    }

    public static void SampleMethodOne()
    {
        Console.WriteLine("SampleMethodOne Invoked");

    }
    public static void SampleMethodTwo()
    {
        Console.WriteLine("SampleMethodTwo Invoked");
    }
    public static void SampleMethodThree()
    {
        Console.WriteLine("SampleMethodThree Invoked");
    }
}
```

```
public delegate void SampleDelegate();

public class Sample
{
    static void Main()
    {
        SampleDelegate del = new SampleDelegate(SampleMethodOne);
        del += SampleMethodTwo;
        del += SampleMethodThree;
        del -= SampleMethodTwo;

        del();
    }

    public static void SampleMethodOne()
    {
        Console.WriteLine("SampleMethodOne Invoked");

    }
    public static void SampleMethodTwo()
    {
        Console.WriteLine("SampleMethodTwo Invoked");
    }
    public static void SampleMethodThree()
    {
        Console.WriteLine("SampleMethodThree Invoked");
    }
}
```

Exception Handling

An exception is an unforeseen error that occurs when a program is running.

Examples:
Trying to read from a file that does not exist, throws FileNotFoundException.
Trying to read from a database table that does not exist, throws a SqlException.

Showing actual unhandled exceptions to the end user is bad for two reasons
1. Users will be annoyed as they are cryptic and does not make much sense to the end users.
2. Exceptions contain information, that can be used for hacking into your application

An exception is actually a class that derives from System.Exception class. The System.Exception class has several useful properties, that provide valuable information about the exception.
Message: Gets a message that describes the current exception
Stack Trace: Provides the call stack to the line number in the method where the exception occurred.

Releasing System Resources

We use try, catch and finally blocks for exception handling:
try - The code that can possibly cause an exception will be in the try block.
catch - Handles the exception.
finally - Clean and free resources that the class was holding onto during the program execution. Finally block is optional.

Note: It is a good practice to always release resources in the finally block, because finally block is guaranteed to execute, irrespective of whether there is an exception or not.

Specific exceptions will be caught before the base general exception, so specific exception blocks should always be on top of the base exception block. Otherwise, you will encounter a compiler error.

Inner Exception

The InnerException property returns the Exception instance that caused the current exception.

To retain the original exception pass it as a parameter to the constructor, of the current exception

Always check if inner exception is not null before accessing any property of the inner exception object, else, you may get Null Reference Exception

To get the type of InnerException use GetType() method

Custom Exception - Steps

1. Create a class that derives from System.Exception class. As a convention, end the class name with Exception suffix. All .NET exceptions end with, exception suffix.

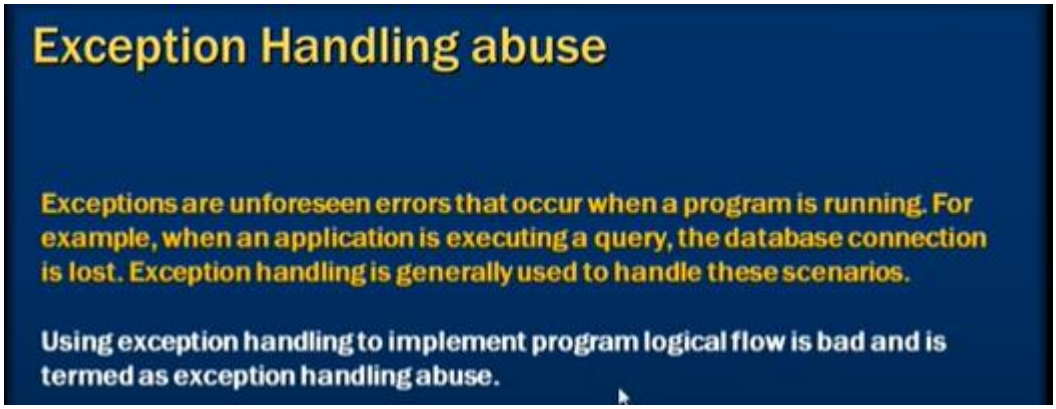
2. Provide a public constructor, that takes in a string parameter. This constructor simply passes the string parameter, to the base exception class constructor.

3. Using InnerExceptions, you can also track back the original exception. If you want to provide this capability for your custom exception class, then overload the constructor accordingly.

4. If you want your Exception class object to work across application domains, then the object must be serializable. To make your exception class serializable mark it with Serializable attribute and provide a constructor that invokes the base Exception class constructor that takes in SerializationInfo and StreamingContext objects as parameters.

***Note:** It is also possible to provide your own custom serialization, which will discuss in a later session.*

```
[Serializable]
public class UserAlreadyLoggedInException : Exception
{
    public UserAlreadyLoggedInException()
        : base()
    {
    }
    public UserAlreadyLoggedInException(string message)
        : base(message)
    {
    }

    public UserAlreadyLoggedInException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    public UserAlreadyLoggedInException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
}
```




```
ndlingAbuse                                                    Main()
{
    Console.WriteLine("Please enter Numerator");
    int Numerator = Convert.ToInt32(Console.ReadLine());

    Console.WriteLine("Please enter Denominator");
    int Denominator = Convert.ToInt32(Console.ReadLine());

    int Result = Numerator / Denominator;

    Console.WriteLine("Result = {0}", Result);
}
catch (FormatException)
{
    Console.WriteLine("Please enter a number");
}
catch (OverflowException)
{
    Console.WriteLine("Only numbers between {0} && {1} are allowed", Int32.MinValue, Int32.MaxValue);
}
catch (DivideByZeroException)
{
    Console.WriteLine("Denominator cannot be zero");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
```

Handling Exception Handling Abuse

```
public static void Main()
{
    try
    {
        Console.WriteLine("Please enter Numerator");
        int Numerator;
        bool IsNumeratorConversionSuccessful = Int32.TryParse(Console.ReadLine(),out Numerator);

        if (IsNumeratorConversionSuccessful)
        {
            Console.WriteLine("Please enter Denominator");
            int Denominator;
            bool IsDenominatorConversionSuccessful = Int32.TryParse(Console.ReadLine(), out Denominator);

            if (IsDenominatorConversionSuccessful && Denominator != 0)
            {
                int Result = Numerator / Denominator;
                Console.WriteLine("Result = {0}", Result);
            }
            else
            {
                if (Denominator == 0)
                {
```

Enums

If a program uses set of integral numbers, consider replacing them with enums, which makes the program more

- Readable
- Maintainable

1. Enums are enumerations.
2. Enums are strongly typed constants. Hence, an explicit cast is needed to convert from enum type to an integral type and vice versa. Also, an enum of one type cannot be implicitly assigned to an enum of another type even though the underlying value of their members are the same.
3. The default underlying type of an enum is int.
4. The default value for first element is ZERO and gets incremented by 1.
5. It is possible to customize the underlying type and values.
6. Enums are value types.
7. Enum keyowrd (all small letteres) is used to create enumerations, where as Enum class, contains static GetValues() and GetNames() methods which can be used to list Enum underlying type values and Names.

```
public enum Gender
{
    Unkown,
    Male,
    Female
}
```

```
    public static void Main()
    {
        short[] Values = (short[])Enum.GetValues(typeof(Gender));

        foreach (short value in Values)
        {
            Console.WriteLine(value);
        }

        string[] Names = Enum.GetNames(typeof(Gender));

        foreach (string Name in Names)
        {
            Console.WriteLine(Name);
        }
    }
}

public enum Gender : short
{
    Unknown = 1,
    Male = 3,
    Female = 23
}
```

```
Gender gender = (Gender)3;
int Num = (int)Gender.Unknown;
```

Types Vs Type Members

In this example Customer is the Type and fields, Properties and method are type members.

So, in general classes, structs, enums, interfaces, delegates are called as types and fields, properties, constructors, methods etc., that normally reside in a type are called as type members.

In C# there are 5 different access modifiers:

1. Private
2. Protected
3. Internal
4. Protected Internal
5. Public

Type members can have all the access modifiers, where as types can have only 2 (internal, public) of the 5 access modifiers

Note: Using regions you can expand and collapse sections of your code either manually, or using visual studio Edit -> Outlining -> Toggle All Outlining

Private, Public & Protected

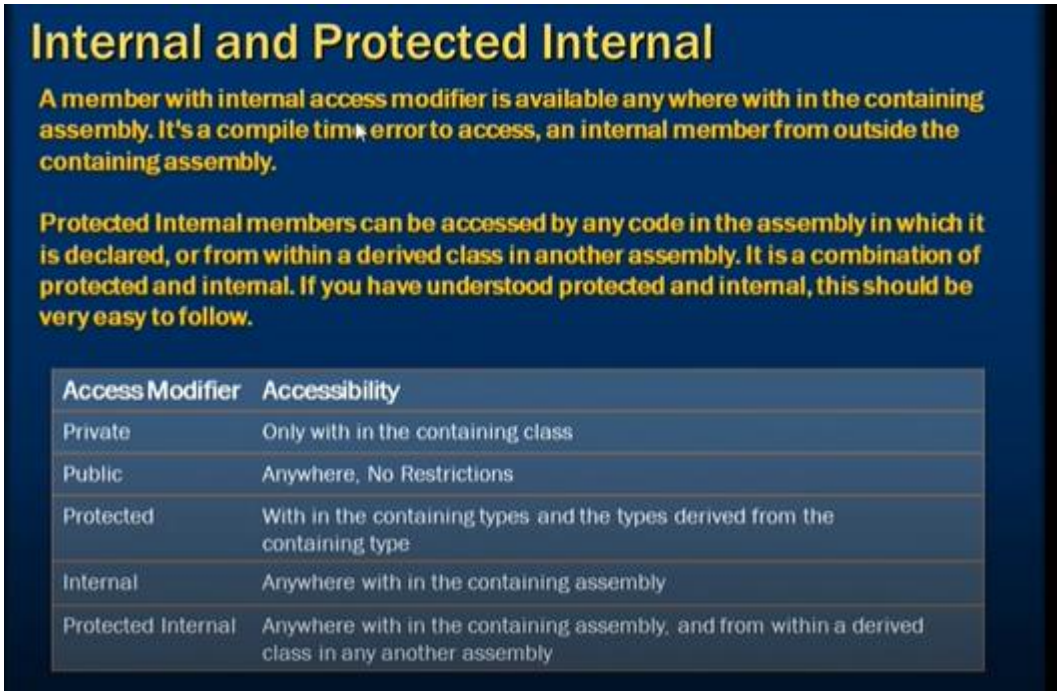
There are 5 different access modifiers in C#.

1. Private
2. Protected
3. Internal
4. Protected Internal
5. Public

Private members are available only with in the containing type, where as public members are available any where. There is no restriction.

Protected Members are available, with in the containing type and to the types that derive from the containing type.

Access Modifier	Accessibility
Private	Only with in the containing class
Public	Any where, No Restrictions
Protected	With in the containing types and the types derived from the containing type



Default Access Modifier for Types➔ internal

Default Access Modifier for Type Members➔ private

Attributes

Attributes allow you to add declarative information to your programs. This information can then be queried at runtime using reflection.

There are several Pre-defined Attributes provided by .NET. It is also possible to create your own Custom Attributes.

A few pre-defined attributes with in the .NET framework:

Obsolete - Marks types and type members outdated
WebMethod - To expose a method as an XML Web service method
Serializable - Indicates that a class can be serialized

It is possible to customize the attribute using parameters

An attribute is a class that inherits from System.Attribute base class.

Reflection

Reflection is the ability of inspecting an assemblies' metadata at runtime. It is used to find all types in an assembly and/or dynamically invoke methods in an assembly.

Uses of reflection:

1. When you drag and drop a button on a win forms or an asp.net application. The properties window uses reflection to show all the properties of the Button class. So, reflection is extensively used by IDE or a UI designers.

2. Late binding can be achieved by using reflection. You can use reflection to dynamically create an instance of a type, about which we don't have any information at compile time. So, reflection enables you to use code that is not available at compile time.

3. Consider an example where we have two alternate implementations of an interface. You want to allow the user to pick one or the other using a config file. With reflection, you can simply read the name of the class whose implementation you want to use from the config file, and instantiate an instance of that class. This is another example for late binding using reflection.

```
private static void Main()
{
    Type T = Type.GetType("Pragim.Customer");
    Console.WriteLine("Full Name = {0}", T.FullName);
    Console.WriteLine("Just the Name = {0}", T.Name);
    Console.WriteLine("Just the Namespace = {0}", T.Namespace);
    Console.WriteLine();

    Console.WriteLine("Properties in Customers");
    PropertyInfo[] properties = T.GetProperties();
    foreach (PropertyInfo property in properties)
    {
        Console.WriteLine(property.PropertyType.Name + " " + property.Name);
    }

    Console.WriteLine();
    Console.WriteLine("Methods in Customers class");
    MethodInfo[] methods = T.GetMethods();
    foreach (MethodInfo method in methods)
    {
        Console.WriteLine(method.ReturnType.Name + " " + method.Name);
    }
```

Early v/s Late Binding

```
Customer C1 = new Customer();
string fullName = C1.GetFullName("Pragim", "Tech");
Console.WriteLine("Full Name = {0}", fullName);
```

```
Assembly executingAssembly = Assembly.GetExecutingAssembly();
Type customerType = executingAssembly.GetType("Pragim.Customer");
object customerInstance = Activator.CreateInstance(customerType);
MethodInfo getFullName = customerType.GetMethod("GetFullName");
string[] methodParameters = new string[2];
methodParameters[0] = "Pragim"; //FirstName
methodParameters[1] = "Tech";   //LastName
string fullName = (string)getFullName.Invoke(customerInstance, methodParameters);
Console.WriteLine("Full Name = {0}", fullName);
```

Difference between early and late binding:

1. Early binding can flag errors at compile time. With late binding there is a risk of run time exceptions.
2. Early binding is much better for performance and should always be preferred over late binding. Use late binding only when working with an objects that are not available at compile time.

Generics

Generics are introduced in C# 2.0. Generics allow us to design classes and methods decoupled from the data types.

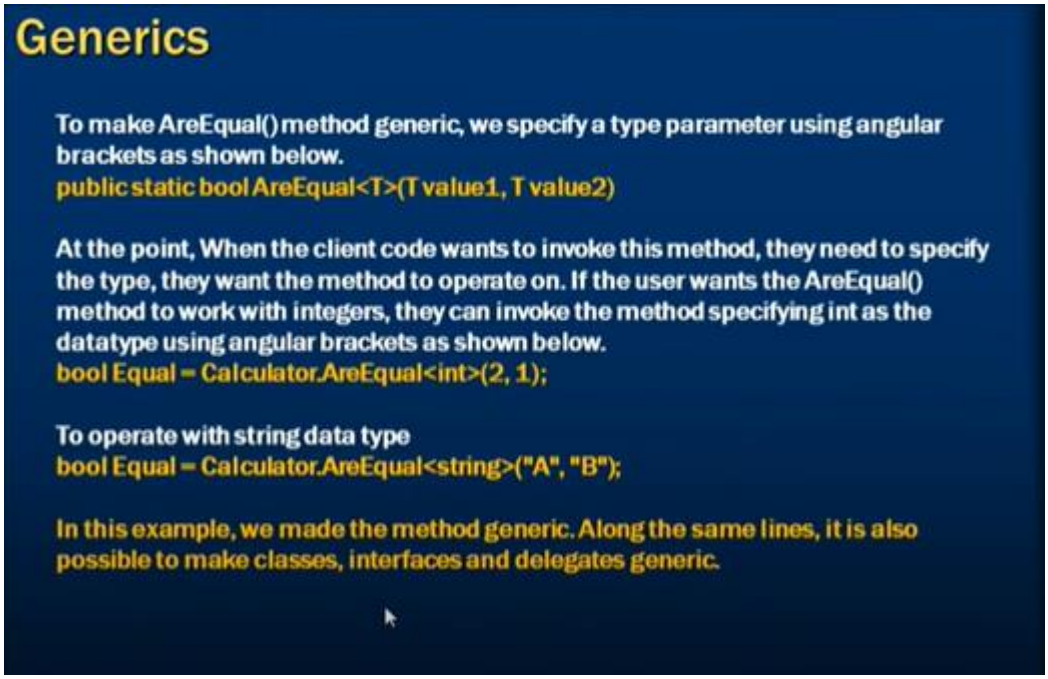
Generic classes are extensively used by collection classes available in System.Collections.Generic namespace. (Covered in the next session)

One way of making AreEqual() method reusable, is to use object type parameters. Since, every type in .NET directly or indirectly inherit from System.Object type, AreEqual() method works with any data type, but the problem is performance degradation due to boxing and unboxing happening.

Also, AreEuqal() method is no longer type safe. It is now possible to pass integer for the first parameter, and a string for the second parameter. It doesn't really make sense to compare strings with integers.

So, the probem with using System.Object type is that

1. AreEqual() method is not type safe
2. Performance degradation due to boxing and unboxing.



There are four methods get Inherited from the *System.Object* to every types

1. GetType (Cannot be extended)
2. ToString
3. Equals
4. GetHashCode

Always recommended to override GetHashCode too if Equals is being overridden.

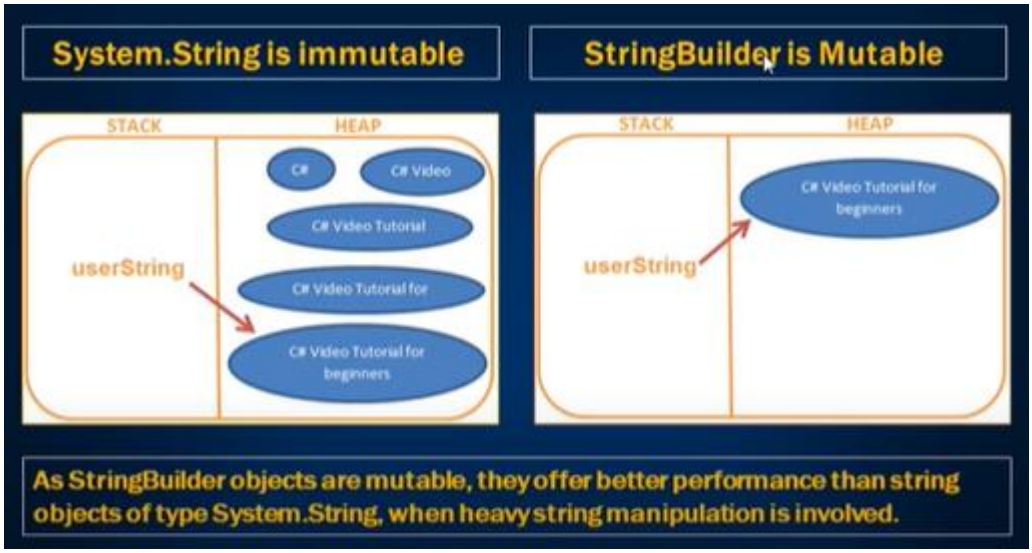
ToString: Should be overridden in every type to provide the string representation of the instance, let say if you have a *Customer* class then ToString of that instance might return Last Name, First Name, Id etc. so you could identify *Customer* Instance right away.

Equals:

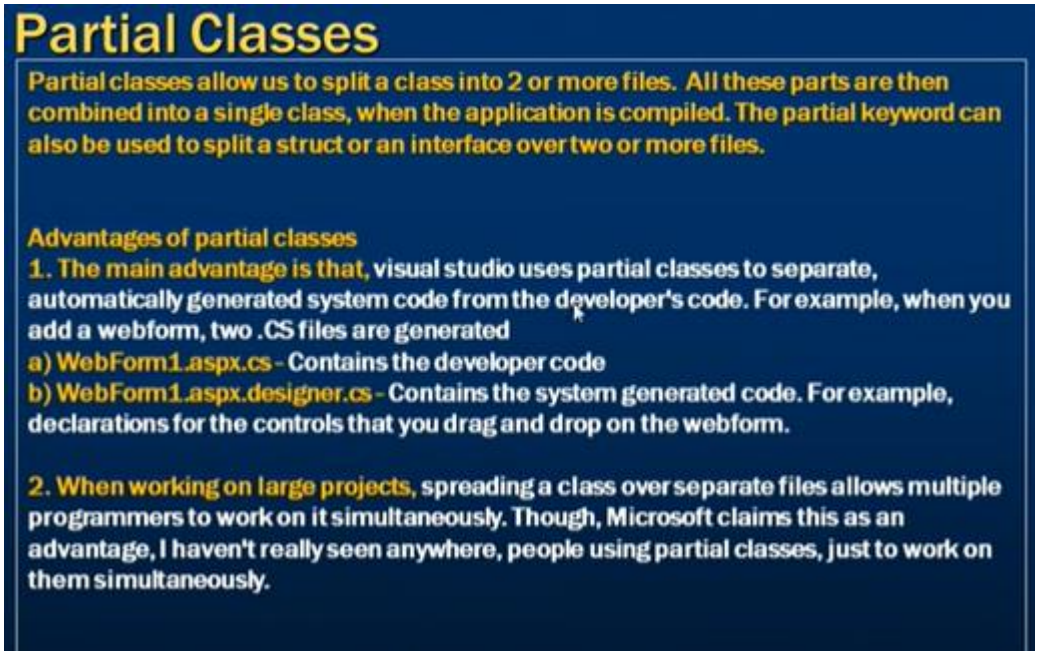
```
public override bool Equals(object obj)
{
    if (obj == null)
    {
        return false;
    }

    if (!(obj is Customer))
    {
        return false;
    }

    return this.FirstName == ((Customer)obj).FirstName &&
        this.LastName == ((Customer)obj).LastName;
}
```

GetHashCode:

```
public override int GetHashCode()
{
    return this.FirstName.GetHashCode() ^ this.LastName.GetHashCode();
}
```



The CLR implements an array to store strings. Arrays are a fixed size data structure, meaning that they cannot be dynamically increased or decreased in size. Once an array is assigned a size, the size cannot be changed. To make an array larger, the data must be copied and cloned into a new array, which is put into a new block of memory by the CLR.



Creating Partial Classes

1. All the parts spread across different files, must use the partial keyword.

2. All the parts spread across different files, must have the same access modifiers.

3. If any of the parts are declared abstract, then the entire type is considered abstract.

4. If any of the parts are declared sealed, then the entire type is considered sealed.

5. If any of the parts inherit a class, then the entire type inherits that class.

6. C# does not support multiple class inheritance. Different parts of the partial class, must not specify different base classes.

7. Different parts of the partial class can specify different base interfaces, and the final type implements all of the interfaces listed by all of the partial declarations.

8. Any members that are declared in a partial definition are available to all of the other parts of the partial class.

Partial Methods

1. A partial class or a struct can contain partial methods.

2. A partial method is created using the partial keyword.

3. A partial method declaration consists of two parts.
i) The definition (only the method signature)
ii) The implementation.
These may be in separate parts of a partial class, or in the same part.

4. The implementation for a partial method is optional. If we don't provide the implementation, the compiler removes the signature and all calls to the method.

5. Partial methods are private by default, and it is a compile time error to include any access modifiers, including private.

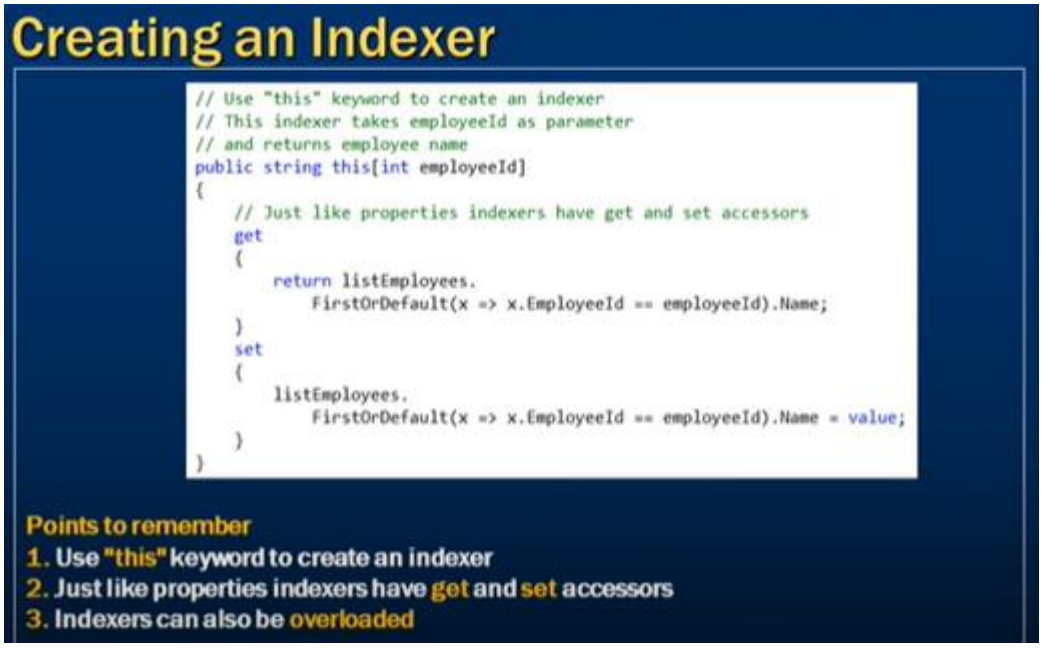
6. It is a compile time error, to include declaration and implementation at the same time for a partial method.

7. A partial method return type must be void. Including any other return type is a compile time error.

8. Signature of the partial method declaration, must match with the signature of the implementation.

9. A partial method must be declared within a partial class or partial struct. A non partial class or struct cannot include partial methods.

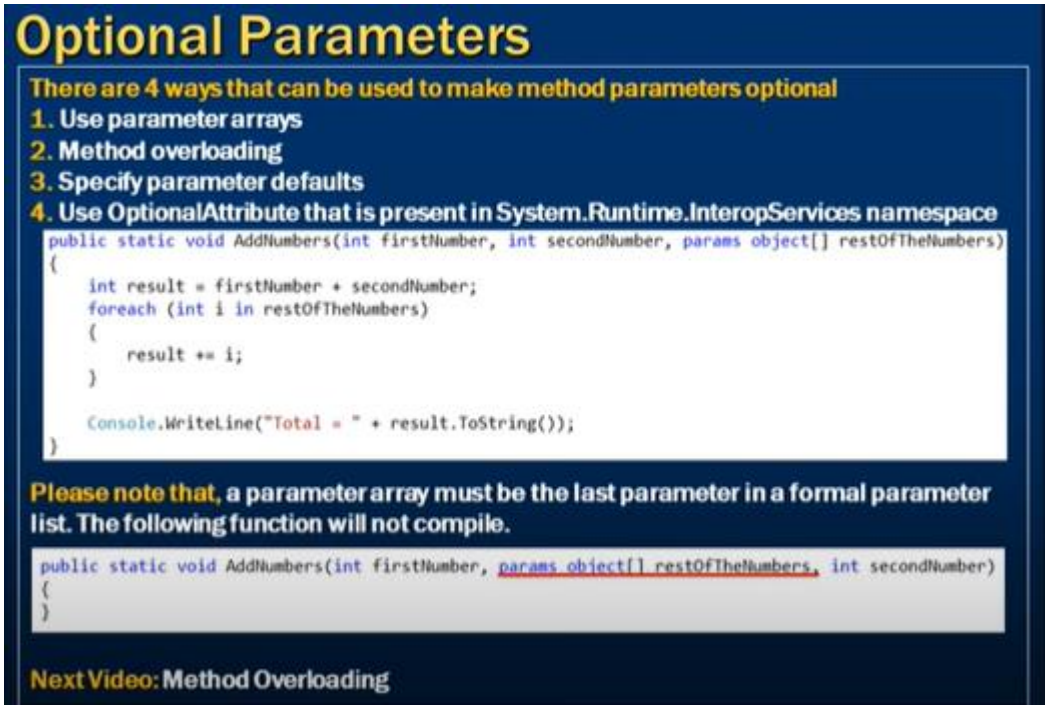
10. A partial method can be implemented only once. Trying to implement a partial method more than once, raises a compile time error



Indexers cannot have return values as void type

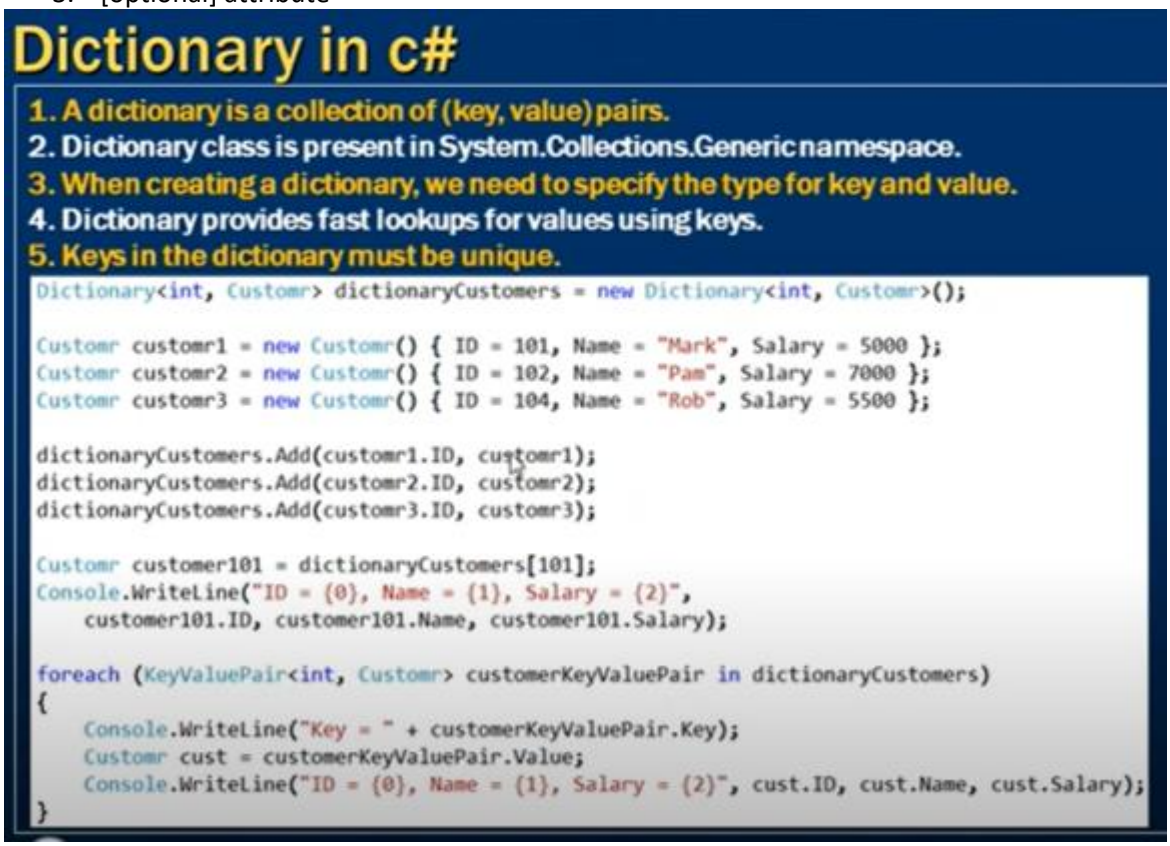
Override Indexers

```
public string this[int employeeId]
{
    get
    {
        return listEmployees.FirstOrDefault(emp => emp.EmployeeId == employeeId).Name;
    }
    set
    {
        listEmployees.FirstOrDefault(emp => emp.EmployeeId == employeeId).Name = value;
    }
}
public string this[int employeeId, int Age]
{
    get
    {
        return listEmployees.FirstOrDefault(emp => emp.EmployeeId == employeeId).Name;
    }
    set
    {
        listEmployees.FirstOrDefault(emp => emp.EmployeeId == employeeId).Name = value;
    }
}
```



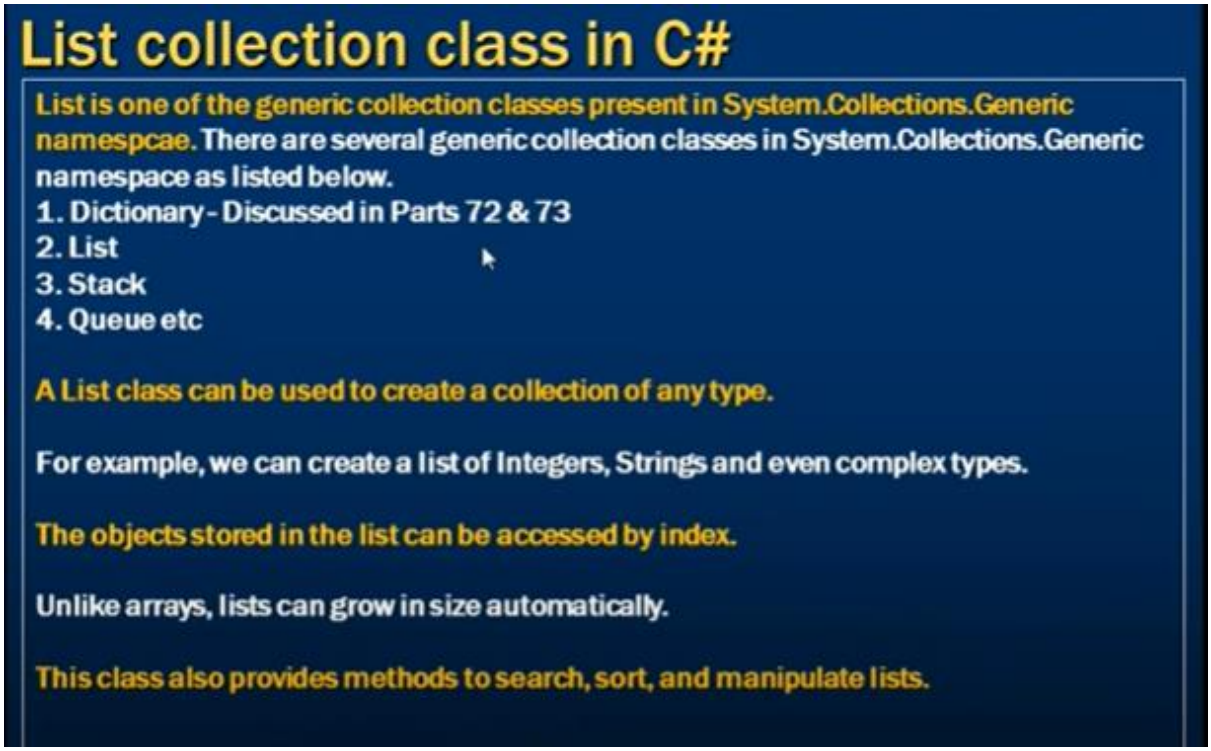
Optional Parameters

1. params int[]
2. default value assignment
3. [optional] attribute

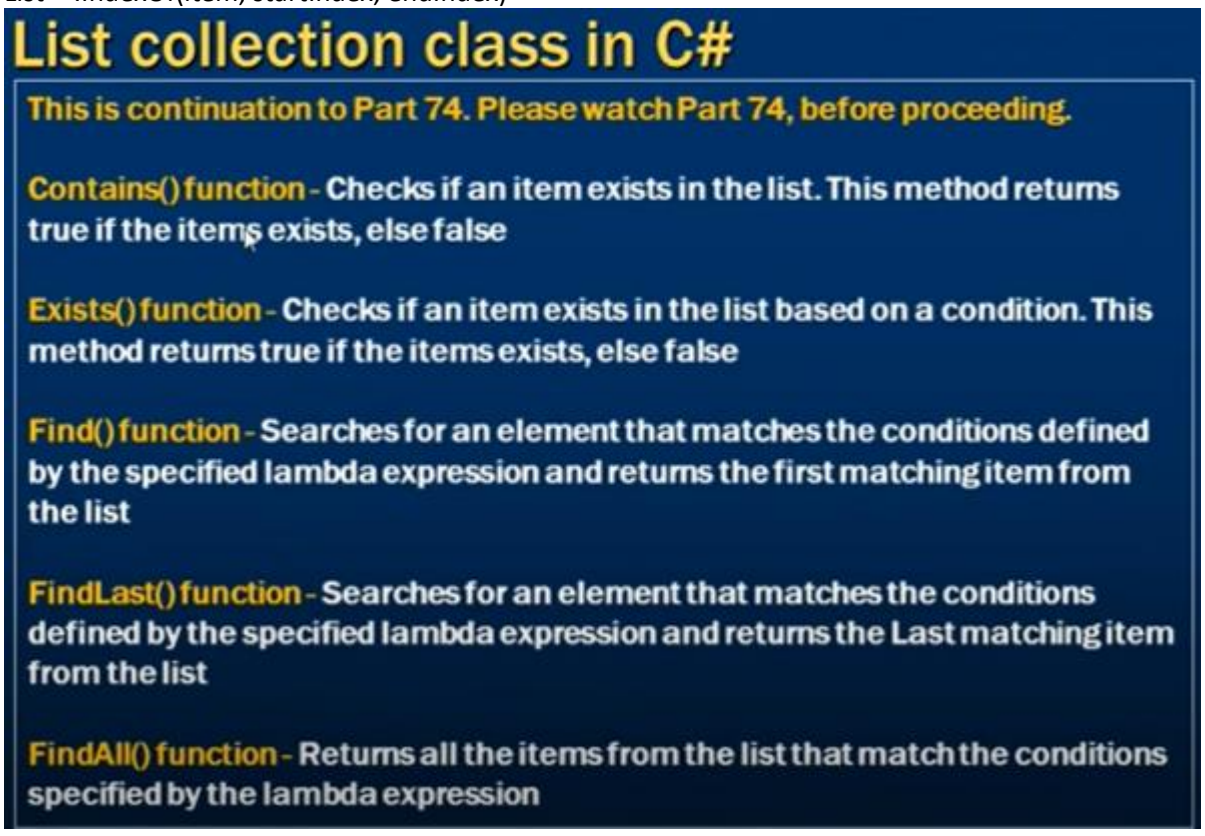


Array to Dictionary

```
Dictionary<int, Customr> dict = customers.ToDictionary(cust => cust.ID, cust => cust);
```



List<>.Insert(index, item)➔ other items after given index will get pushed one index down
List<>.IndexOf(item)
List<>.IndexOf(item, startIndex)
List<>.IndexOf(item, startIndex, endIndex)



FindIndex() function - Returns the index of the first item, that matches the condition specified by the lambda expression. There are 2 other overloads of this method which allows us to specify the range of elements to search, with in the list.

FindLastIndex() function - Returns the index of the last item, that matches the condition specified by the lambda expression. There are 2 other overloads of this method which allows us to specify the range of elements to search, with in the list.

Convert an array to a List - Use ToList() method

Convert a list to an array - Use ToArray() method

Convert a List to a Dictionary - Use ToDictionary() method

Working with Generic list class & Ranges

Please watch Part 74 & 75 before proceeding with this video

AddRange() - Add() method allows you to add one item at a time to the end of the list, where as AddRange() allows you to add another list of items, to the end of the list.

GetRange() - Using an item index, we can retrieve only one item at a time from the list, if you want to get a list of items from the list, then use GetRange() function. This function expects 2 parameters, i.e the start index in the list and the number of elements to return.

InsertRange() - Insert() method allows you to insert a single item into the list at a specificed index, where as InsertRange() allows you, to insert another list of items to your list at the specified index.

RemoveRange() - Remove() function removes only the first matching item from the list. RemoveAt() function, removes the item at the specified index in the list. RemoveAll() function removes all the items that matches the specified condition. RemoveRange() method removes a range of elements from the list. This function expects 2 parameters, i.e the start index in the list and the number of elements to remove. If you want to remove all the elements from the list without specifying any condition, then use Clear() function.

Sorting a list of simple types

Please watch Part 76 before proceeding with this video
http://www.youtube.com/user/kudvenkat/videos?view=1

Sorting a list of simple types like int, string, char etc, is straight forward. Just invoke the sort() method on the list instance and the data will be automatically sorted in ascending order.

```
List<int> numbers = new List<int> { 1, 8, 7, 5, 2, 3, 4, 9, 6 };
numbers.Sort();
```

If you want the data to be retrieved in descending order, use Reverse() method on the list instance.

```
numbers.Reverse();
```

However, when you do the same thing on a complex type like Customer, we get a runtime invalid operation exception - Failed to compare 2 elements in the array. This because, .NET runtime does not know, how to sort complex types. We have to tell the way we want data to be sorted in the list by implementing IComparable interface.

How is the sort functionality working for simple types like int, string, char etc?
That is because these types (int, string, decimal, char etc) have implemented IComparable interface already.

Sorting a list of complex types

To sort a list of complex types without using LINQ, the complex type has to implement IComparable interface and provide implementation for CompareTo() method. CompareTo() method returns an integer, and the meaning of the return value is shown below. RETURN VALUE is

Greater than ZERO - The current instance is greater than the object being compared with.
Less than ZERO - The current instance is less than the object being compared with.
is ZERO - The current instance is equal to the object being compared with.

Alternatively you can also invoke CompareTo() method. Salary property of the Customer object is int. CompareTo() method is already implemented on integer type, so we can invoke this method and return it's value.

```
public class Customer : IComparable<Customer>
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Salary { get; set; }

    public int CompareTo(Customer obj)
    {
        //if (this.Salary > obj.Salary)
        //    return 1;
        //else if (this.Salary < obj.Salary)
        //    return -1;
        //else
        //    return 0;

        return this.Salary.CompareTo(obj.Salary);
    }
}
```

If you prefer not to use the Sort functionality provided by the class, then you can provide your own, by implementing IComparer interface. For example, if I want the customers to sorted by name instead of salary.

Step 1: Implement IComparer interface

```
public class SortByName : IComparer<Customer>
{
    public int Compare(Customer x, Customer y)
    {
        return x.Name.CompareTo(y.Name);
    }
}
```

Step 2: Pass an instance of the class that implements IComparer interface, as an argument to the Sort() method.

```
SortByName sortByName = new SortByName();
listCutomers.Sort(sortByName);
```

Using Comparison delegate with List<T>

Please watch Part 78 from C# Tutorial video series, before proceeding.
http://www.youtube.com/user/kudvenkat/videos?view=1

One of the overloads of the Sort() method in List class expects Comparison delegate to be passed as an argument

```
public void Sort(Comparison<T> comparison);
```

Approach 1:

Step 1: Create a function whose signature matches the signature of System.Comparison delegate. This is the method where we need to write the logic to compare 2 customer objects.

```
private static int CompareCustomers(Customer c1, Customer c2)
{
    return c1.ID.CompareTo(c2.ID);
}
```

Step 2: Create an instance of System.Comparison delegate, and then pass the name of the function created in Step1 as the argument. So, at this point "Comparison" delegate is pointing to our function that contains the logic to compare 2 customer objects.

```
Comparison<Customer> customerComparer = new Comparison<Customer>(CompareCustomers);
```

Step 3: Pass the delegate instance as an argument, to Sort() function.

```
listCutomers.Sort(customerComparer);
```

Approach 2:

In Approcah1 this is what we have done

1. Created a private function that contains the logic to compare customers
2. Created an instance of Comparison delegate, and then passed the name of the private function to the delegate.
3. Finally passed the delegate instance to the Sort() method.

Do we really have to follow all these steps. Isn't there any other way?

The above code can be simplified using delegate keyword as shown below.

```
listCutomers.Sort(delegate(Customer c1, Customer c2)
{
    return (c1.ID.CompareTo(c2.ID));
});
```

Approach 3: The code in Approach 2, can be further simplified using lambda expression as shown below.

```
listCutomers.Sort((c1, c2) => c1.ID.CompareTo(c2.ID));
```

Useful methods of List collection class

Please watch Part 79 from C# Tutorial video series, before proceeding.
http://www.youtube.com/user/kudvenkat/videos?view=1

In this video, we will discuss the following methods

TrueForAll() - Returns true or false depending on whether if every element in the list matches the conditions defined by the specified predicate.

AsReadOnly() - Returns a read-only wrapper for the current collection. Use this method, if you don't want the client code to modify the collection i.e add or remove any elements from the collection. The ReadOnlyCollection will not have methods to add or remove items from the collection. You can only read items from this collection.

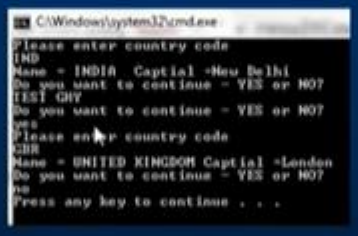
TrimExcess() - Sets the capacity to the actual number of elements in the List, if that number is less than a threshold value.

According to MSDN:
This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large List<T> can be considerable. So the TrimExcess method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain. The current threshold is 90 percent, but this could change in the future.

When to use a Dictionary over List

Please watch Part 80 from C# Tutorial video series, before proceeding.
http://www.youtube.com/user/kudvenkat/videos?view=1

Find() method of the List class loops thru each object in the list until a match is found. So, if you want to lookup a value using a key dictionary is better for performance over list. So, use dictionary when you know the collection will be primarily used for lookups.



Queue is a generic FIFO (First In First Out) collection class that is present in System.Collections.Generic namespace. The Queue collection class is analogous to a queue at the ATM machine to withdraw money. The order in which people queue up, will be the order in which they will be able to get out of the queue and withdraw money from the ATM. The Queue collection class operates in a similar fashion. The first item to be added (enqueued) to the queue, will be the first item to be removed (dequeued) from the Queue.

To add items to the end of the queue, use Enqueue() method.

To remove an item that is present at the beginning of the queue, use Dequeue() method.

A foreach loop iterates thru the items in the queue, but will not remove them from the queue.

To check if an item, exists in the queue, use Contains() method.

What is the difference between Dequeue() and Peek() methods?
Dequeue() method removes and returns the item at the beginning of the queue, where as Peek() returns the item at the beginning of the queue, without removing it.

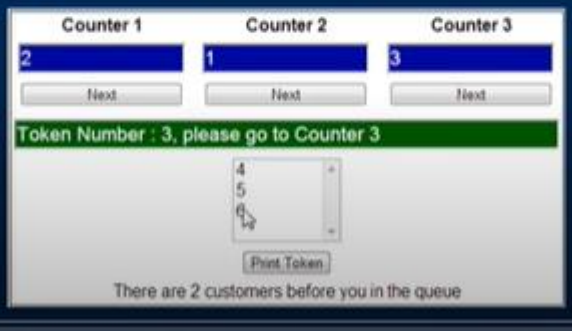
Real time example - Queue

When you walk into a bank or a passport office, you will collect a token and wait in the queue for your token number to be called.

From the application perspective, when a token is issued, the token number will be added to the end of the Queue.

When a representative at the counter is available to server a customer, he will push the "Next" button and the token number that is present at the beginning of the queue, will be dequeued.

So, this is one example, where a Queue collection class can be effectively used.



Generic Stack collection class

Stack is a generic LIFO (Last In First Out) collection class that is present in System.Collections.Generic namespace. The Stack collection class is analogous to a stack of plates. If you want to add a new plate to the stack of plates, you place it on top of all the already existing plates. If you want to remove a plate from the stack, you will first remove the one that you have last added. The stack collection class also operates in a similar fashion. The last item to be added (pushed) to the stack, will be the first item to be removed (popped) from the stack.

To insert an item at the top of the stack, use Push() method.

To remove and return the item that is present at the top of the stack, use Pop() method.

A foreach loop iterates thru the items in the stack, but will not remove them from the stack. The items from the stack are retrieved in LIFO (Last In First Out), order. The last element added to the Stack is the first one to be returned.

To check if an item exists in the stack, use Contains() method.

What is the difference between Pop() and Peek() methods?
Pop() method removes and returns the item at the top of the stack, where as Peek() returns the item at the top of the stack, without removing it.

Real time example - Stack

Two common scenarios, where a stack can be used
1. Implementing UNDO functionality
2. Implementing browser back button

Multithreading in C#

Link for C#, ASP.NET, SQL Server, WCF & MVC tutorial
http://www.youtube.com/user/kudvenkat/playlists

What is a Process:
Process is what the operating system uses to facilitate the execution of a program by providing the resources required. Each process has a unique process Id associated with it. You can view the process within which a program is being executed using windows task manager.

What is Thread:
Thread is a light weight process. A process has at least one thread which is commonly called as main thread which actually executes the application code. A single process can have multiple threads.

Please Note: All the threading related classes are present in System.Threading namespace.

Advantages & Disadvantages

Link for C#, ASP.NET, SQL Server, WCF & MVC tutorial
http://www.youtube.com/user/kudvenkat/playlists

Advantages of multithreading:

1. To maintain a responsive user interface
2. To make efficient use of processor time while waiting for I/O operations to complete
3. To split large, CPU-bound tasks to be processed simultaneously on a machine that has multiple processors/cores

Disadvantages of multithreading:

1. On a single processor/core machine threading can affect performance negatively as there is overhead involved with context-switching
2. Have to write more lines of code to accomplish the same task
3. Multithreaded applications are difficult to write, understand, debug and maintain

Please Note: Only use multithreading when the advantages of doing so outweigh the disadvantages.

ThreadStart Delegate

Link for C#, ASP.NET, SQL Server, WCF & MVC tutorial
http://www.youtube.com/user/kudvenkat/playlists

```
class Program
{
    public static void Main()
    {
        Thread T1 = new Thread(Number.PrintNumbers);
        T1.Start();
    }
}

class Number
{
    public static void PrintNumbers()
    {
        for (int i = 1; i <= 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

To create a THREAD, create an instance of Thread class and to it's constructor pass the name of the function that we want the thread to execute.

ThreadStart Delegate

```
Thread T1 = new Thread(Number.PrintNumbers);
T1.Start();
```

We can rewrite the above line using ThreadStart delegate as shown below

```
Thread T1 = new Thread(new ThreadStart(Number.PrintNumbers));
T1.Start();
```

Why a delegate need to be passed as a parameter to the Thread class constructor?

The purpose of creating a Thread is to execute a function. A delegate is a type safe function pointer, meaning it points to a function that the thread has to execute. In short, all threads require an entry point to start execution. Any thread you create will need an explicitly defined entry point i.e a pointer to the function where they should begin execution. So threads always require a delegate.

In the code below, we are not explicitly creating the ThreadStart delegate, then how is it working here?

```
Thread T1 = new Thread(Number.PrintNumbers);
T1.Start();
```

It's working in spite of not creating the ThreadStart delegate explicitly because the framework is doing it automatically for us.

We can also rewrite the same line using delagate() keyword

```
Thread T1 = new Thread(delegate() { Number.PrintNumbers(); });
```

The same line rewritten using lambda expression

```
Thread T1 = new Thread(() => Number.PrintNumbers());
```

```
class Program
{
    public static void Main()
    {
        Number number = new Number();
        Thread T1 = new Thread(number.PrintNumbers);
        T1.Start();
    }
}

class Number
{
    public void PrintNumbers()
    {
        for (int i = 1; i <= 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

Thread function need not be a static function always. It can also be a non-static function. In case of non-static function we have to create an instance of the class.

ParameterizedThreadStart Delegate

```
class Number
{
    public static void PrintNumbers(object target)
    {
        int number = 0;
        if (int.TryParse(target.ToString(), out number))
        {
            for (int i = 1; i <= number; i++)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Use ParameterizedThreadStart delegate to pass data to the thread function

```
class Program
{
    public static void Main()
    {
        Console.WriteLine("Please enter the target number");
        object target = Console.ReadLine();

        ParameterizedThreadStart parameterizedThreadStart =
            new ParameterizedThreadStart(Number.PrintNumbers);
        Thread T1 = new Thread(parameterizedThreadStart);
        T1.Start(target);
    }
}
```

ParameterizedThreadStart Delegate

Please note: Using ParameterizedThreadStart delegate and Thread.Start(Object) method to pass data to the Thread function is not type safe as they operate on object datatype and any type of data can be passed.

If you try to change the data type of the target parameter of PrintNumbers() function from object to int, a compiler error will be raised as the signature of PrintNumbers() function does not match with the signature of ParameterizedThreadStart delegate.

Next Video: Passing data to the Thread function without loosing the type saftey

Passing data to the Thread function

```
class Number
{
    int _target;
    public Number(int target)
    {
        this._target = target;
    }

    public void PrintNumbers()
    {
        for (int i = 1; i <= _target; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

To pass data to the Thread function in a type safe manner, encapsulate the thread function and the data it needs in a helper class and use the ThreadStart delegate to execute the thread function.

Next Video: Retrieving data from Thread function using callback method

```
class Program
{
    public static void Main()
    {
        Console.WriteLine("Please enter the target number");
        int target = Convert.ToInt32(Console.ReadLine());

        Number number = new Number(target);
        Thread T1 = new Thread(new ThreadStart(number.PrintNumbers));
        T1.Start();
    }
}
```

Callback method to get data from thread

Example:

```
Please enter the target number
5
Sum of numbers is 15
```

Main thread retrieves the target number from the user.

Main thread creates a child thread and pass the target number to the child thread.

The child thread computes the sum of numbers and then returns the sum to the Main thread using callback function.

The callback method prints the sum of numbers

Step 1: Create a callback delegate. The actual callback method signature should match with the signature of this delegate.

```
public delegate void SumOfNumbersCallback(int sumOfNumbers);
```

Step 2: Create a helper class to compute the sum of numbers and to call the callback method.

```
class Number
{
    int _target;
    SumOfNumbersCallback _callbackMethod;

    public Number(int target, SumOfNumbersCallback callbackMethod)
    {
        this._target = target;
        this._callbackMethod = callbackMethod;
    }

    public void ComputeSumOfNumbers()
    {
        int sum = 0;
        for (int i = 1; i <= _target; i++)
        {
            sum = sum + i;
        }
        if (_callbackMethod != null)
            _callbackMethod(sum);
    }
}
```

Thread.Join & Thread.IsAlive functions

Join blocks the current thread and makes it wait until the thread on which Join method is invoked completes. Join method also has a overload where we can specify the timeout. If we don't specify the timeout the calling thread waits indefinitely, until the thread on which Join() is invoked completes. This overloaded Join(int millisecondsTimeout) method returns boolean. True if the thread has terminated otherwise false.

Join is particularly useful when we need to wait and collect result from a thread execution or if we need to do some clean-up after the thread has completed.

IsAlive returns boolean. True if the thread is still executing otherwise false.

```
namespace ThreadingExample
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("Main Started");
            Thread T1 = new Thread(Program.Thread1Function);
            T1.Start();

            Thread T2 = new Thread(Program.Thread2Function);
            T2.Start();

            T1.Join();
            Console.WriteLine("Thread1Function Completed");

            T2.Join();
            Console.WriteLine("Thread1Function Completed");
```

Protecting Shared Resources

What happens if shared resources are not protected from concurrent access in multithreaded program?

The output or behaviour of the program can become inconsistent if the shared resources are not protected from concurrent access in multithreaded program

```
class Program
{
    static int Total = 0;
    public static void Main()
    {
        AddOneMillion();
        AddOneMillion();
        AddOneMillion();
        Console.WriteLine("Total = " + Total);
    }

    public static void AddOneMillion()
    {
        for (int i = 1; i <= 1000000; i++)
        {
            Total++;
        }
    }
}
```

This program is a single-threaded program. In the Main() method, AddOneMillion() method is called 3 times, and it updates the Total field correctly as expected, and finally prints the correct total i.e 3M.

Protecting Shared Resources

Using Interlocked.Increment() method: Increments a specified variable and stores the result, as an atomic operation.

```
public static void AddOneMillion()
{
    for (int i = 1; i <= 1000000; i++)
    {
        Total++;
    }
}
```

```
public static void AddOneMillion()
{
    for (int i = 1; i <= 1000000; i++)
    {
        Interlocked.Increment(ref Total);
    }
}
```

Locking:

```
static object _lock = new object();

public static void AddOneMillion()
{
    for (int i = 1; i <= 1000000; i++)
    {
        lock (_lock)
        {
            Total++;
        }
    }
}
```

Protecting Shared Resources

Which option is better?

From a performance perspective using Interlocked class is better over using locking. Locking locks out all the other threads except a single thread to read and increment the Total variable. This will ensure that the Total variable is updated safely. The downside is that since all the other threads are locked out, there is a performance hit.

The Interlocked class can be used with addition/subtraction (increment, decrement, add, etc.) on and int or long field. The Interlocked class has methods for incrementing, decrementing, adding, and reading variables atomically.

Monitor v/s lock

Both Monitor class and lock provides a mechanism that synchronizes access to objects. lock is the shortcut for Monitor.Enter with try and finally

```
static object _lock = new object();
public static void AddOneMillion()
{
    for (int i = 1; i <= 1000000; i++)
    {
        lock (_lock)
        {
            Total++;
        }
    }
}
```

==

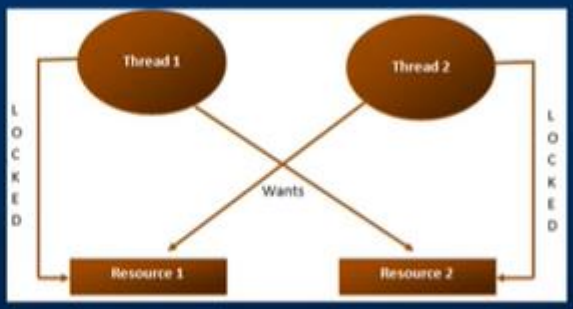
```
static object _lock = new object();
public static void AddOneMillion()
{
    for (int i = 1; i <= 1000000; i++)
    {
        Monitor.Enter(_lock);
        try
        {
            Total++;
        }
        finally
        {
            Monitor.Exit(_lock);
        }
    }
}
```

Deadlocks

Link for C#, ASP.NET, SQL Server, WCF & MVC tutorials
http://www.youtube.com/user/kudvenkat/videos?view=1

When a deadlock can occur
Let's say we have 2 threads
a) Thread 1
b) Thread 2

and 2 resources
a) Resource 1
b) Resource 2



Thread 1 has already acquired a lock on Resource 1 and wants to acquire a lock on Resource 2. At the same time Thread 2 has already acquired a lock on Resource 2 and wants to acquire a lock on Resource 1.

Two threads never give up their locks, hence a deadlock

Deadlocks

```
public class AccountManager
{
    Account _fromAccount; Account _toAccount; double _amountToTransfer;
    public AccountManager(Account fromAccount, Account toAccount, double amountToTransfer)
    {
        this._fromAccount = fromAccount;
        this._toAccount = toAccount;
        this._amountToTransfer = amountToTransfer;
    }
    public void Transfer()
    {
        lock (_fromAccount)
        {
            Thread.Sleep(1000);
            lock (_toAccount)
            {
                _fromAccount.Withdraw(_amountToTransfer);
                _toAccount.Deposit(_amountToTransfer);
            }
        }
    }
}
```

```
public static void Main()
{
    Console.WriteLine("Main Started");
    Account accountA = new Account(101, 5000);
    Account accountB = new Account(102, 3000);

    AccountManager accountManagerA = new AccountManager(accountA, accountB, 1000);
    Thread T1 = new Thread(accountManagerA.Transfer);
    T1.Name = "T1";

    AccountManager accountManagerB = new AccountManager(accountB, accountA, 2000);
    Thread T2 = new Thread(accountManagerB.Transfer);
    T2.Name = "T2";

    T1.Start();
    T2.Start();

    T1.Join();
    T2.Join();
    Console.WriteLine("Main Completed");
}
```

Resolving Deadlocks

```
public void Transfer()
{
    lock (_fromAccount)
    {
        Thread.Sleep(1000);
        lock (_toAccount)
        {
            _fromAccount.Withdraw(_amount
            _toAccount.Deposit(_amountTo
        }
    }
}
```

```
public void Transfer()
{
    object _lock1, _lock2;
    if (_fromAccount.ID < _toAccount.ID)
    {
        _lock1 = _fromAccount; _lock2 = _toAccount;
    }
    else
    {
        _lock1 = _toAccount; _lock2 = _fromAccount;
    }
    lock (_lock1)
    {
        Thread.Sleep(1000);
        lock (_lock2)
        {
            _fromAccount.Withdraw(_amountToTransfer);
            _toAccount.Deposit(_amountToTransfer);
        }
    }
}
```

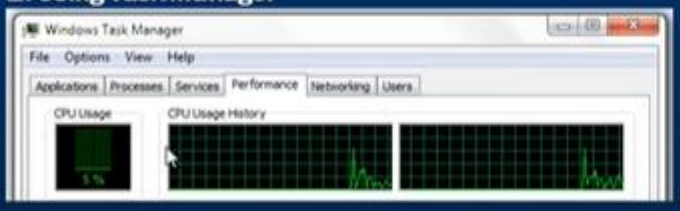
```
AccountManager accountManagerA = new AccountManager(accountA, accountB, 1000);
Thread T1 = new Thread(accountManagerA.Transfer);

AccountManager accountManagerB = new AccountManager(accountB, accountA, 2000);
Thread T2 = new Thread(accountManagerB.Transfer);
```

Performance Implications

Link for C#, ASP.NET, SQL Server, WCF & MVC tutorials
http://www.youtube.com/user/kudvenkat/videos?view=1

How to find out how many processors you have on your machine

1. Using Task Manager



2. Use the following code in any .net application

```
Environment.ProcessorCount
```

3. On the windows command prompt window, type the following

```
echo %NUMBER_OF_PROCESSORS%
```

Performance Implications

On a machine that has multiple processors, multiple threads can execute application code in parallel on different cores. For example, if there are two threads and two cores, then each thread would run on an individual core. This means, performance is better.

If two threads take 10 milli-seconds each to complete, then on a machine with 2 processors, the total time taken is 10 milli-seconds.

On a machine that has a single processor, multiple threads execute, one after the other or wait until one thread finishes. It is not possible for a single processor system to execute multiple threads in parallel. Since the operating system switches between the threads so fast, it just gives us the illusion that the threads run in parallel. On a single core/processor machine multiple threads can affect performance negatively as there is overhead involved with context-switching.

If two threads take 10 milli-seconds each to complete, then on a machine with 1 processor, the total time taken is
20 milli-seconds + (Thread context switching time, if any)

Anonymous Methods

What is an anonymous method?

Anonymous method is a method without a name. Introduced in C# 2, they provide us a way of creating delegate instances without having to write a separate method.

```
// Step 1: Create a method whose signature matches
// with the signature of Predicate<Employee> delegate
private static bool FindEmployee(Employee Emp)
{
    return Emp.ID == 102;
}

// Step 2: Create an instance of Predicate<Employee> delegate and pass the
// method name as an argument to the delegate constructor
Predicate<Employee> predicateEmployee = new Predicate<Employee>(FindEmployee);

// Step 3: Now pass the delegate instance as the argument for Find() method
Employee employee = listEmployees.Find(x => predicateEmployee(x));
Console.WriteLine("ID = {0}, Name {1}", employee.ID, employee.Name);

// Anonymous method is being passed as an argument to the Find() method
// This anonymous method replaces the need for Step 1, 2 and 3
employee = listEmployees.Find(delegate(Employee x) { return x.ID == 102; });
Console.WriteLine("ID = {0}, Name {1}", employee.ID, employee.Name);
```

Non - Blocking Example – async & await

```
private int CountCharacters()
{
    int count = 0;
    using (StreamReader reader = new StreamReader("C:\\Data\\Data.txt"))
    {
        string content = reader.ReadToEnd();
        count = content.Length;
        Thread.Sleep(5000);
    }
    return count;
}

private async void btnProcessFIle_Click(object sender, EventArgs e)
{
    Task<int> task = new Task<int>(CountCharacters);
    task.Start();

    lblCount.Text = "Processing file. Please wait...";
    int count = await task;
    lblCount.Text = count.ToString() + " characters in file";
}
```