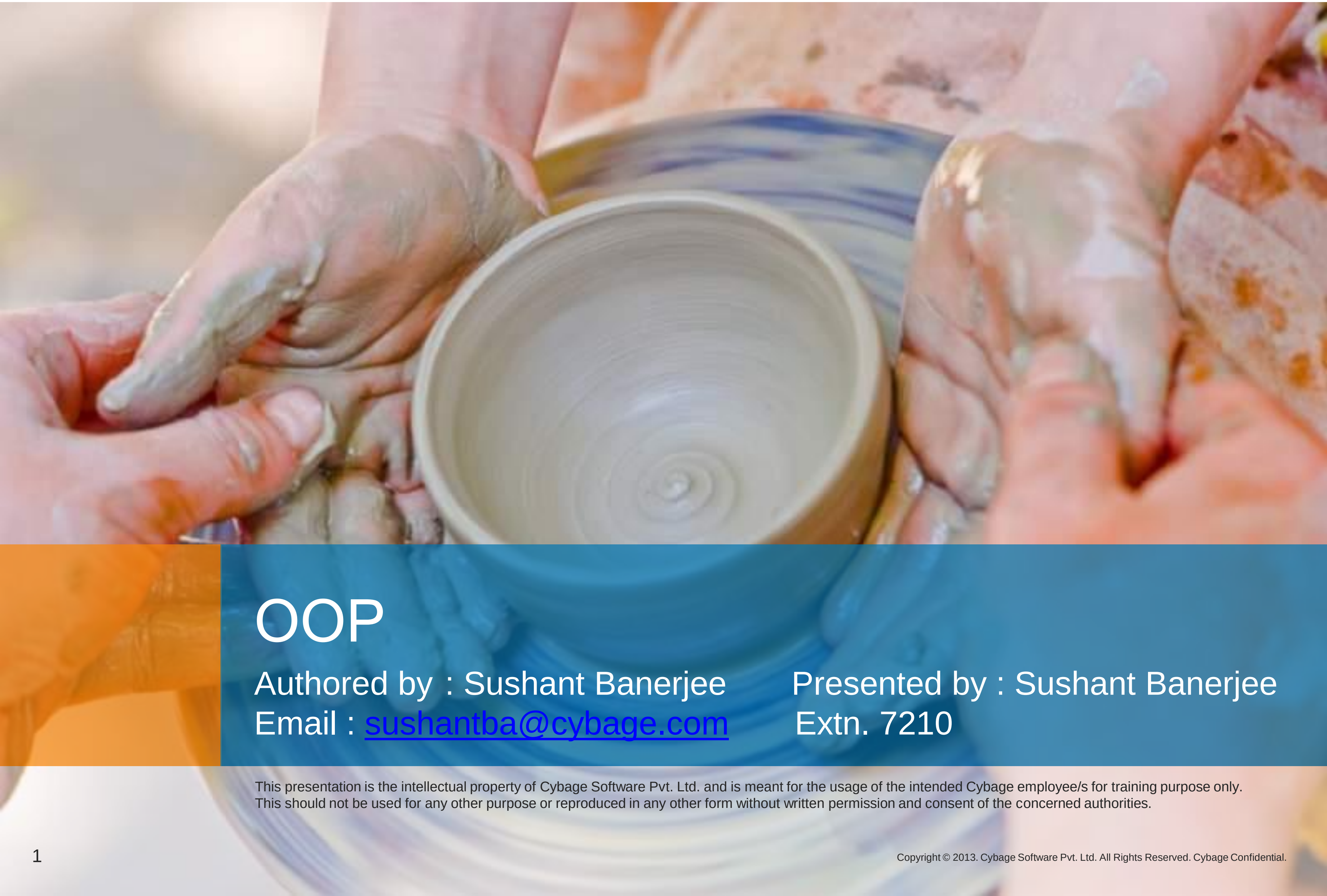


OOP

Authored by : Sushant Banerjee
Email : sushantba@cybage.com

Presented by : Sushant Banerjee
Extn. 7210

This presentation is the intellectual property of Cybage Software Pvt. Ltd. and is meant for the usage of the intended Cybage employee/s for training purpose only. This should not be used for any other purpose or reproduced in any other form without written permission and consent of the concerned authorities.

Agenda

- Properties
- Indexers
- Inheritance
- Polymorphism
- Partial Class and Methods
- Abstract Class and Interface
- Structs and Enums
- System.Object members

Using Properties

```
    class Customer
    {
        private int customerId;
        public int CustomerId
        {
            get { return customerId; }
            set { customerId = value; }
        }
    }
 Customer cust = new Customer();
 cust.CustomerId = 123;
 Console.WriteLine("Customer Id : {0}", cust.CustomerId);
```

Restricting Accessor Accessibility

```
public int CustomerId
{
 get { return customerId; } //public because the property
   itself is public
 protected set { customerId = value; }//only derived class
   object can change the value
}
```

Auto-Implemented Property (C# 3.0)

- Auto-implemented properties can be used when you do not need additional logic to be implemented in the property accessor.
- Auto-implemented properties do not require you to declare private fields to be used by the property accessor.
- The compiler automatically creates private anonymous backing fields that can be accessed by get and set accessors.

```

    class Customer
    {
        public int CustomerId { get; set; }
    }
```

Indexers

```
 class SampleCollection
    {
        private string[] City = new string[100];
        public string this[int i]
        {
            get { return City[i]; }
            set { City[i] = value; }
        }
    }
SampleCollection myCollection = new SampleCollection();
myCollection[0] = "Ahmedabad";
myCollection[1] = "Gandhinagar";
```

Properties Vs. Indexers

Properties	Indexers
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A get accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit value parameter.	A set accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.

Inheritance

```
    class A
    {
        public void MethodA() { }
    }
    class B : A
    {
        public void MethodB()
        {
            B obj = new B();
            obj.MethodA();
        }
    }
```

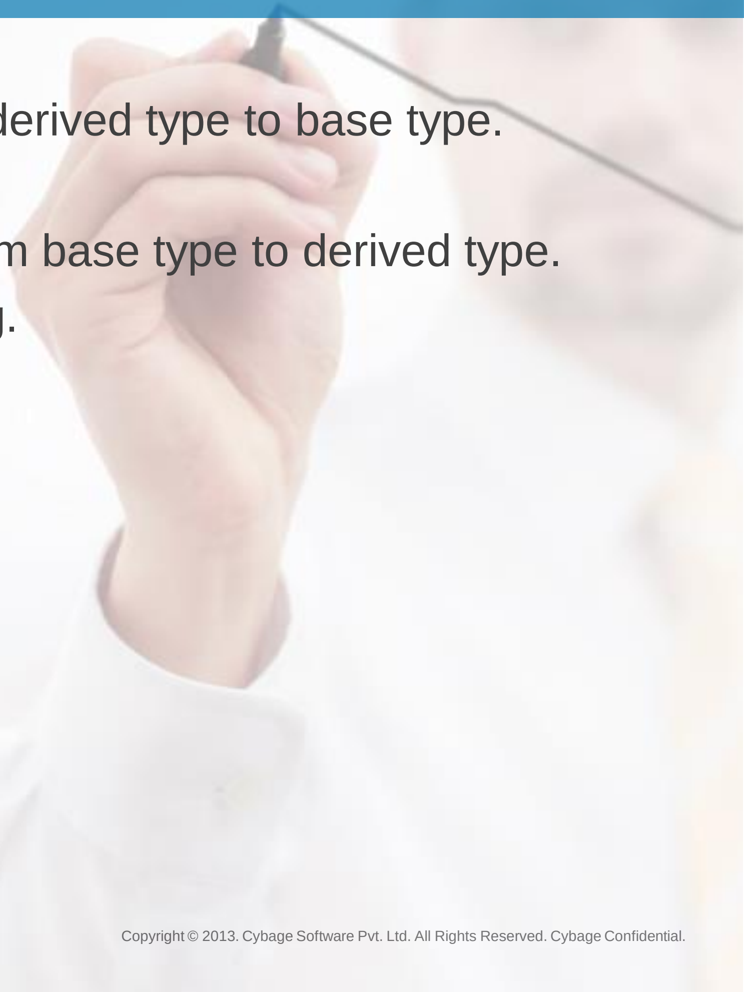
Inheritance (contd…)

```
    class C : B
    {
        static void Main(string[] args)
        {
            C obj = new C();
            obj.MethodB();
            Console.ReadLine();
        }
    }
```

Polymorphism

```
class Shape{
        public virtual void Draw()
        {
            Console.WriteLine("Drawing a Shape");
        }
    }
class Circle : Shape{
        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
            base.Draw();
        }
    }
```

Up Casting and Down Casting



- Up casting happens when cast from derived type to base type.
- Up casting is implicit.
- Down casting happens when cast from base type to derived type.
- Down casting requires explicit casting.

Sealed Classes

- A sealed class prevent derivation.
- For the same reason a sealed class cannot be used as either a base class or an abstract class.

```
//this class can not be derived
public sealed class B : A
{
}
//this line will never compile
public class C : B
{
}
```

Sealed Methods

- A Sealed Method prevents overriding.
- To stop overriding you need to put “sealed” keyword just before “override” keyword.

```
    class B : A
    {
 //This method is no more overridable by the derived class of B
        public sealed override void MethodA()
        {
            Console.WriteLine("Do some work");
        }
    }
```

Overriding ToString Method

- By default the ToString method returns string representation of an object.
- You can override this method in your custom class to return a customized string representation of the type.

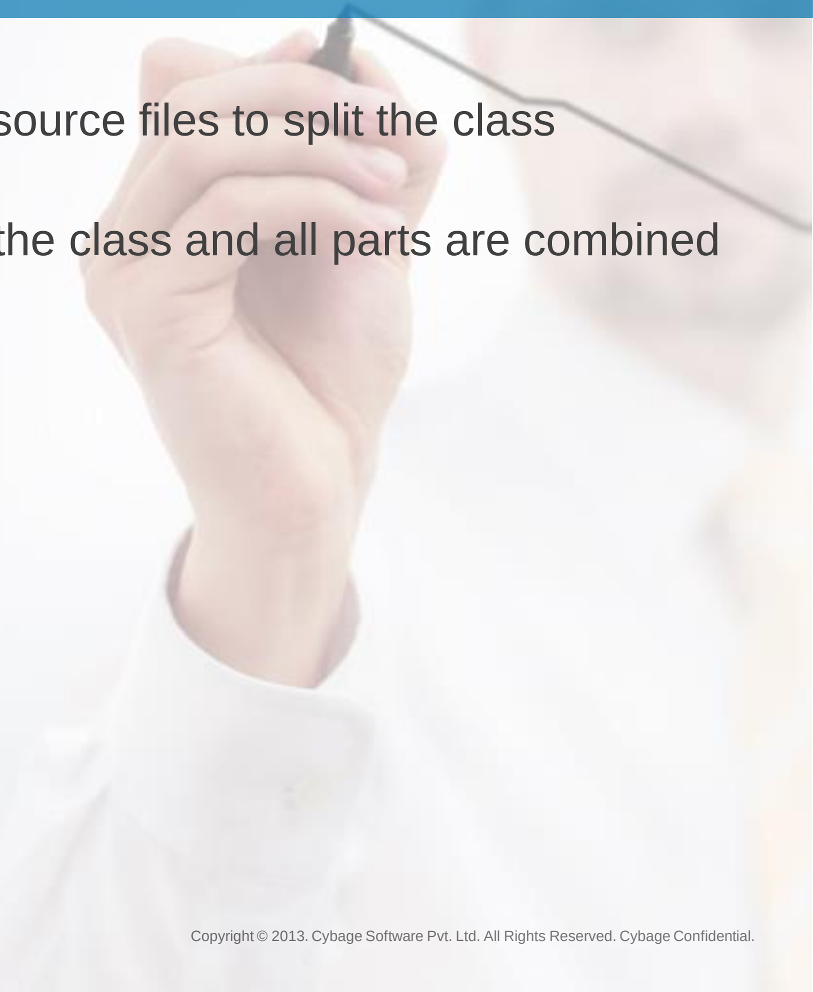
```
class Student
{
    public string Name { get; set; }
    public int StudentId { get; set; }
    public override string ToString()
    {
  return string.Format("Name:{0} and Id:{1}", Name, StudentId);
    }
 }
```

Static Classes

- A static class cannot be instantiated.
- A static class is sealed, so cannot be a base class.
- A static class cannot contain any instance member, you can only define static members in a static class.

```
public static class Student
{
    public static void GetScore(int studentId;)
    {
    }
}
Student.GetScore(121); //use class name to access static
  members
```

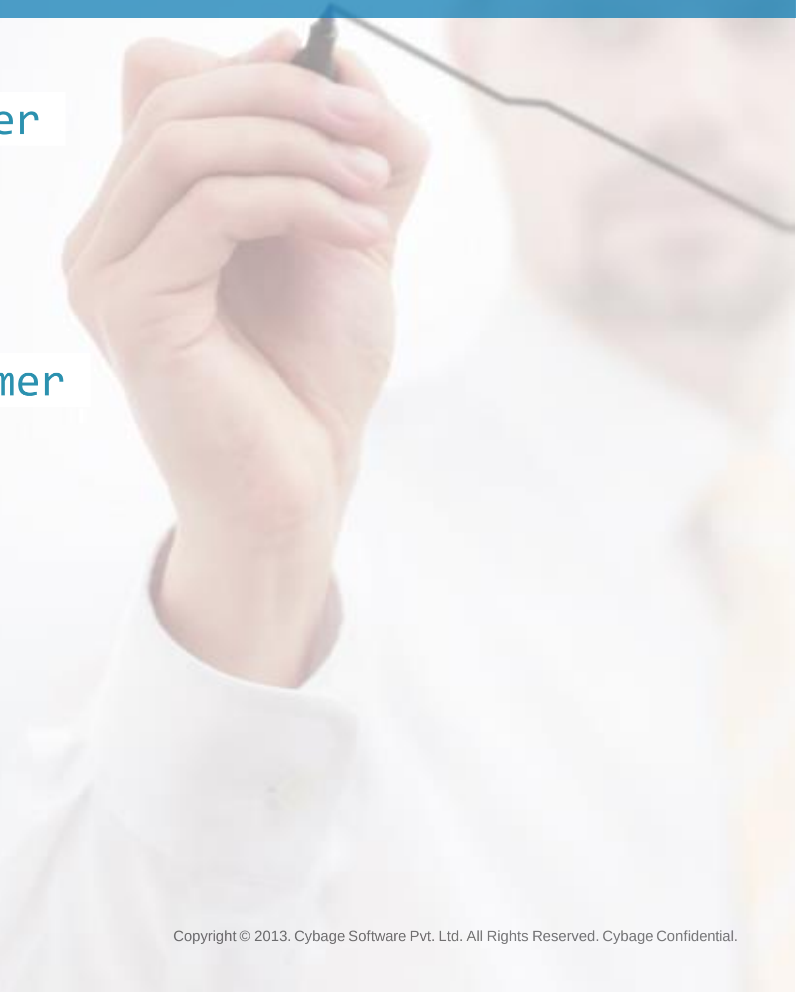
Partial Class



- Partial class uses more than one source files to split the class definition.
- Each source file contain a part of the class and all parts are combined when the application is compiled.

How to Write Partial Classes

```
   public partial class Customer
    {
        //write behavior here
    }
    public partial class Customer
    {
        //write behavior here
    }
```



When to use Partial Class

- Keeping separate files may be useful for the large projects, because it allows multiple developers to work on the same class.
- Visual Studio uses partial class to add auto-generated code to different files, for example one for designing and the other for coding.

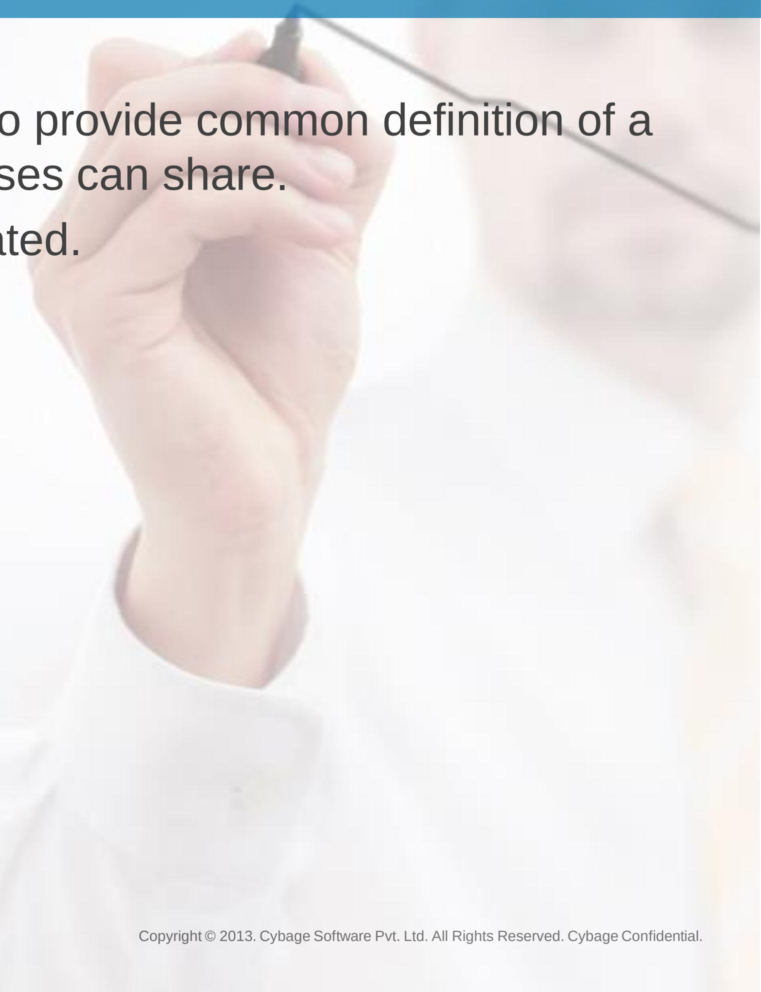
Partial Methods

- A partial class may contain partial methods.
- You can keep partial method definition in one part of the class and optional implementation of the partial method in another part of the class or in a different class.
- A partial method are implicitly private and can’t take any access modifiers.
- Return type of a partial method must be void.
- A partial method can be implemented only once.
- Partial methods support “ref” keyword but not “out” modifier.
- Partial methods do not support virtual, abstract, override and sealed modifiers.

How to Use Partial Methods

```
    //partial Class
    public partial class Customer{
        //partial method definition
        partial void PlaceOrder();

        //partial method implementaion
        partial void PlaceOrder(){
        //write implementation here or it will throw exception
            throw new NotImplementedException();
        }
    }
```

Abstract Class



- The purpose of an abstract class is to provide common definition of a base class that multiple derived classes can share.
- An abstract class cannot be instantiated.

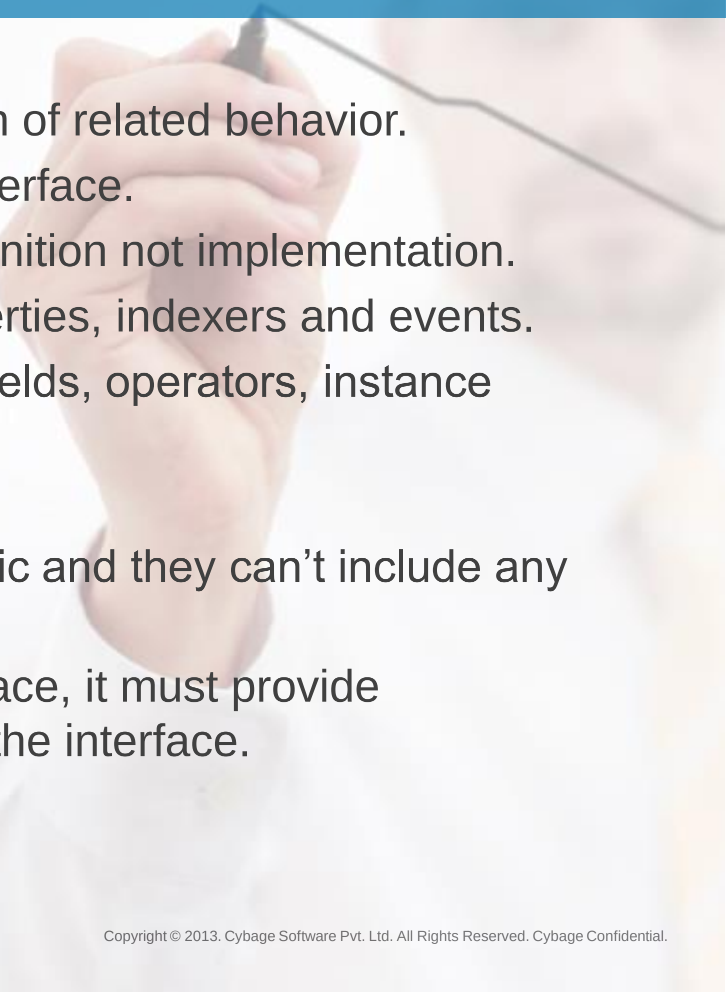
```
   public abstract class A
    {
         //define members here
    }
```

Abstract Methods

- Abstract classes can have abstract methods.
- An abstract method has no implementation.
- Method definition is followed by semicolon instead of method block.
- If you derive an abstract class you must implement all the abstract methods defined in that class.

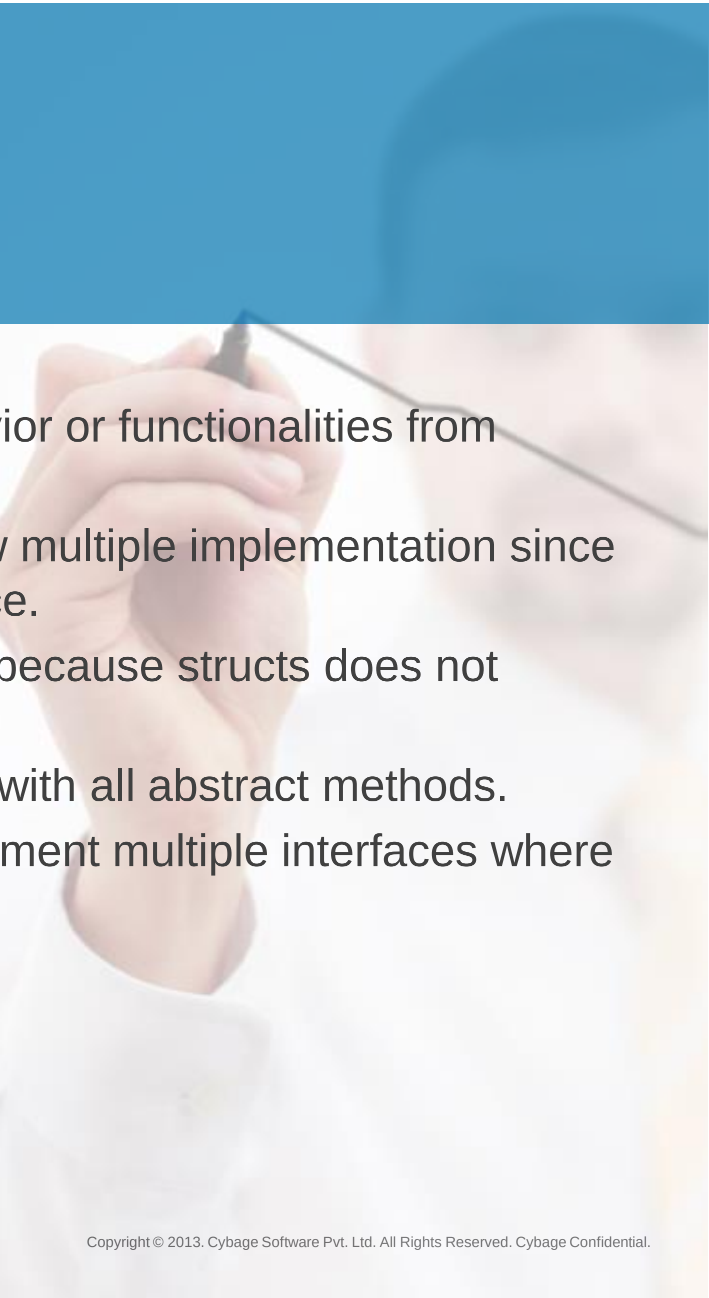
```
public abstract class A
{
    public abstract void MethodA();
}
public class B : A
{
  public override void MethodA() {//define implementation here}
}
```

What is Interface



- An interface can contain only definition of related behavior.
- A class or struct can implement the interface.
- An interface includes only method definition not implementation.
- Interfaces can contain methods, properties, indexers and events.
- An interface can’t contain constants, fields, operators, instance constructors, destructors or types.
- Interface members can not be static.
- Interface members are by default public and they can’t include any access modifiers.
- If a class or struct implement an interface, it must provide implementation of all the members of the interface.

Why Interface



- An interface allow you to include behavior or functionalities from multiple interfaces in your class.
- So, the purpose of interfaces is to allow multiple implementation since C# does not support multiple inheritance.
- Interfaces are useful in case of structs because structs does not support inheritance.
- Interface is similar to an abstract class with all abstract methods.
- The difference is that a class can implement multiple interfaces where a class can inherit from a single class.

How to use Interface

```
interface IShape
{
    void Draw();
}
interface IPaint
{
    void FillColor();
}
```

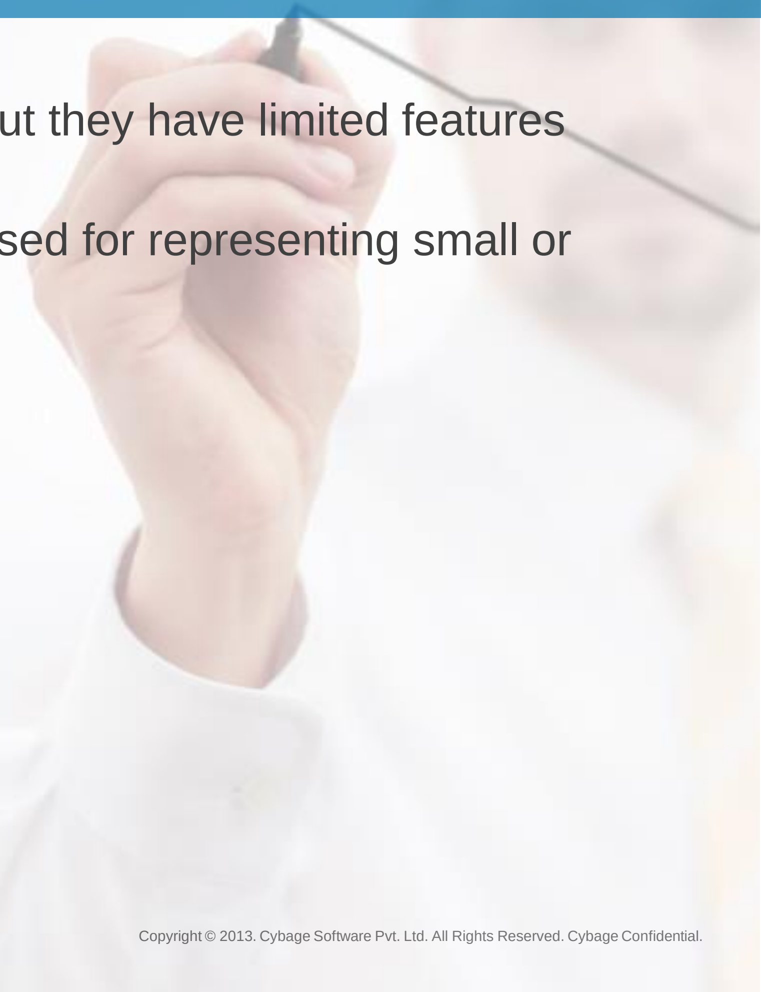
How to use Interface

```
  class Shape : IShape, IPaint{
    //must implement all the methods
    public void Draw(){
        Console.WriteLine("Drawing a Shape");
    }

    public void FillColor(){
      Console.WriteLine("Filling with blue color");
    }
}
```

Structs

- Structs look very similar to classes but they have limited features compare to classes.
- Structs are value types and ideally used for representing small or lightweight objects.

```
    struct ContactInfo
     {
          //add members here
     }
```



Constructors in Struct

- Structs do not support default parameter less constructor, only constructor with parameter is allowed.

```
struct ContactInfo
{
    public string city;
    public long phone;
    public ContactInfo(string ct, long ph)
    {
        city = ct;
        phone = ph;
    }
}
```

Instantiating Structs

- You can create object of a struct either using new operator or without new operator.
- When you use new operator appropriate constructor is called.
- In case you don’t use new operator, no constructor is called.

```
//declaring an object does not initialize members
//because no constructor is called
ContactInfo ci;
ContactInfo cinfo = new ContactInfo();//fields will not
initialize because no default constructor is allowed
ContactInfo cinfo2 = new ContactInfo("Gandhinagar", 123);
//fields will now initialize successfully
```

Structs Vs. Classes

- Structs do not support parameter less constructor.
- You can not initialize an instance field directly inside a struct, you have to initialize only using parameterized constructor.
- Structs do not support inheritance, however structs inherit from the base class object.
- However structs can implement interface just like classes.
- Structs are value types and classes are reference types.

Enums

- An enum is another value type that can be used to define a set of named integral constants that may be assigned to a variable.
- The keyword enum is used to define an enumeration type.
- By default each element in the enum is int.
- By default the first enumerator has the value 0 and the value of each element is increased by 1.

```
//declaring an enum
enum Color { Blue, White, Green };

//Color enum enumerators are Blue = 0, White = 1 and
  Green = 2
```

Why enums

- Let’s consider the example of an enum defined in the .NET Framework Library which allows you to change the color of Console.

```
//using inbuilt enum ConsoleColor
Console.BackgroundColor = ConsoleColor.Cyan;
```

- If it allows to hard code values then there may be some invalid values assigned by the client code results in exception.
- It is better to use enum instead of hard coding values.

Using Enums

```
//declaring an enum
enum Color { Blue, White, Green };

//assigning value of enum element to enum variable
Color favouriteColor = Color.Blue;
//displaying enum value
Console.WriteLine("Favourite Color : {0}",
 favouriteColor);
//displaying enum integral constant
Console.WriteLine("Favourite Color Number : {0}",
 (int)favouriteColor);
```

Changing Default Behavior of enums

- The default underlying type of enum is int, however you can specify types such as byte, sbyte short, ushort, uint, long, ulong.

```
//changing underlying type
enum Color : byte {Blue, White, Green};
```

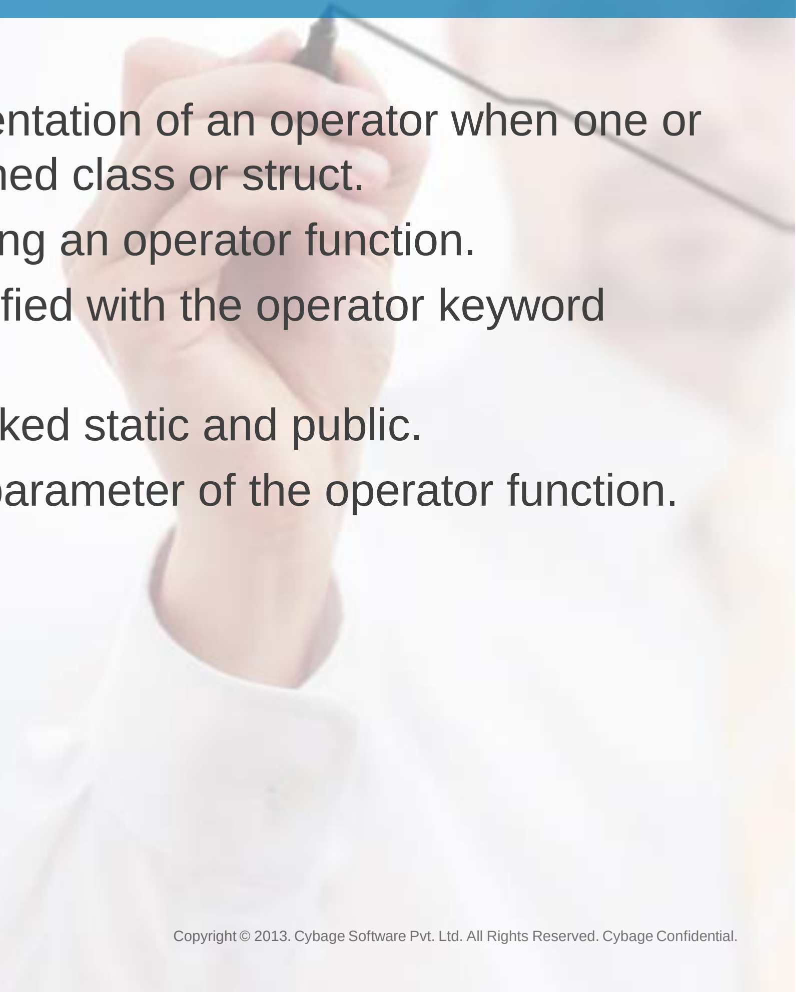
- The default enumerator value starts with 0 and incremented by 1, you can change value for specific elements or all of them.

```
//changing default enumerator values
enum Color { Blue = 1, White = 3, Green = 5 };
```

Advantages of using enums

- You can create a collection of values using an enum and expose the valid values to client code.
- While assigning values of enum Visual Studio IntelliSense shows the defined values that can be used.

Operator Overloading



- You can provide your own implementation of an operator when one or both of the operands are user defined class or struct.
- An operator can be overloaded using an operator function.
- An operator function name is specified with the operator keyword followed by an operator symbol.
- The operator function must be marked static and public.
- Operands can be specified in the parameter of the operator function.

System.Object Class Members

- Object.GetType() Method
- Object.ToString() Method
- Object.Equals() Method
- Object.Finalize() Method
- Object.GetHashCode() Method

Object.GetType()

- This method returns the type of the current instance.

```
int x = 5;
float f = 2.5F;
double d = 3.5D;
Console.WriteLine(x.GetType());//returns System.Int32
Console.WriteLine(f.GetType());//returns System.Single
Console.WriteLine(d.GetType());//returns System.Double
```

Object.ToString() Method

- This method returns a string representation of a current object.
- By default ToString method returns the fully qualified name of the type of the current object.
- You should override the method to add custom code so that the method returns string suitable for display.

Object.Equals()

- Determines whether the specified object is equal to the current object.
- The return type of this method is boolean.
- It returns true if the specified object is equal to the current object, otherwise it returns false.
- The type of comparison depends on whether the current instance is a reference type or a value type.
- In case the current object is reference type, the method tests for reference equality.
- If the current object is value type then it tests for value equality.

Object.Equals()

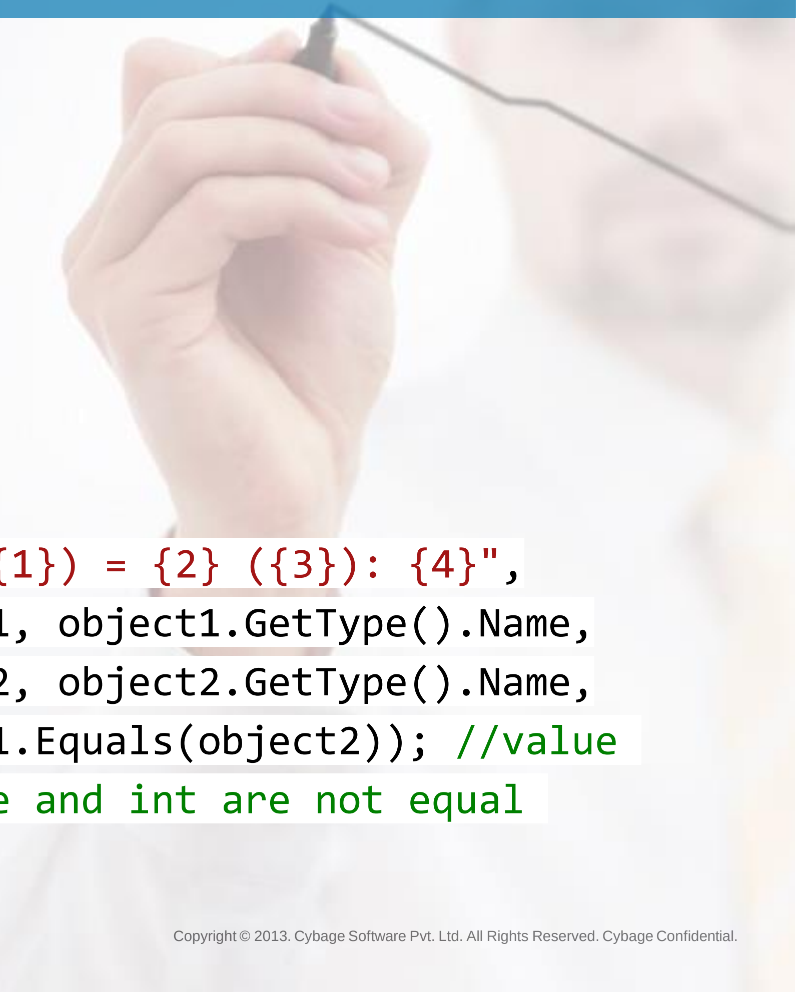
```
public class Employee{
    public string empName;

    public Employee(string name)
    {
        this.empName = name;
    }
}
Employee emp1 = new Employee("Sushant");
Employee emp2 = emp1;

Employee emp3 = new Employee("Sushant");
```

Object.Equals() (contd…)

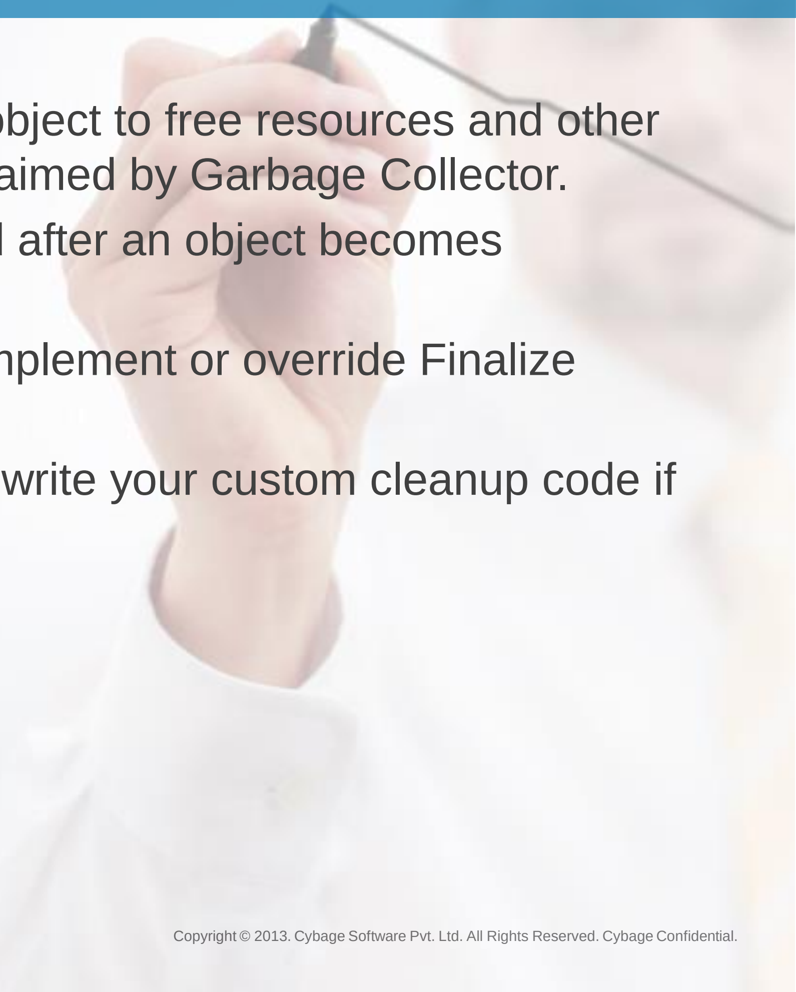
```
//returns true because emp1 and emp2 are equal, since they
  reference same object
Console.WriteLine("emp1 and emp2 = {0}", emp1.Equals(emp2));
//the other way
Console.WriteLine("emp1 and emp2 = {0}", object.Equals(emp1,
  emp2));

//returns false because emp1 and emp3 are not equal, since they
  reference different object although they have the same value.
Console.WriteLine("emp1 and emp3 = {0}", emp1.Equals(emp3));
```

Object.Equals() (contd…)

```
 public static void Main()
    {
        byte value1 = 12;
        int value2 = 12;

        object object1 = value1;
        object object2 = value2;

        Console.WriteLine("{0} ({1}) = {2} ({3}): {4}",
                          object1, object1.GetType().Name,
                          object2, object2.GetType().Name,
                          object1.Equals(object2)); //value
euqality returns false since byte and int are not equal
    }
```



Object.Finalize()



- Object.Finalize method allows an object to free resources and other cleanup operations before it is reclaimed by Garbage Collector.
- This method is automatically called after an object becomes inaccessible.
- C# does not allow you to directly implement or override Finalize method.
- You can make use of destructor to write your custom cleanup code if you are using unmanaged code.

Object.GetHashCode()

- A hash code is numeric value that is used to identify an object during equality testing.
- It can also serve as an index for an object in a collection.
- The GetHashCode method is suitable for use in hashing algorithms and data structures such as a hash table.
- You can override this method and implement your own logic for the hash function and return hash by calling GetHashCode method.

Object.GetHashCode() (contd…)

```
public class Test
{
//overriding GetHashCode method
    public override int GetHashCode()
    {
         //implement your own hash logic
    }
}
Test t = new Test();
Console.WriteLine("The hash value returned by hash function is
  : {0}", t.GetHashCode());
```

Bibliography, Important Links

- C# 5.0 in a Nutshell – Published by O’Reilly
- www.msdn.com – Library
- http://en.wikipedia.org

Any Questions?

Thank you!