Abstract class

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

```
abstract class ShapesClass
{
   abstract public int Area();
}
class Square : ShapesClass
{
   int side = 0;

   public Square(int n)
   {
      side = n;
   }
   // Area method is required to avoid
   // a compile-time error.
   public override int Area()
   {
      return side * side;
   }

   static void Main()
   {
      Square sq = new Square(12);
      Console.WriteLine("Area of the square = {0}", sq.Area());
   }

// Output: Area of the square = 144
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier because the two modifers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the abstract modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.
- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

  ```
  public abstract void MyMethod();
  ```

  The implementation is provided by an overriding methodoverride (C# Reference), which is a member of a non-abstract class.

- It is an error to use the static or virtual modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the abstract modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

In this example, the class DerivedClass is derived from an abstract class BaseClass. The abstract class contains an abstract method, AbstractMethod, and two abstract properties, X and Y.

```
abstract class BaseClass   // Abstract class
{
   protected int _x = 100;
   protected int _y = 150;
   public abstract void AbstractMethod();   // Abstract method
   public abstract int X    { get; }
   public abstract int Y    { get; }
}
```

```
class DerivedClass : BaseClass
{
   public override void AbstractMethod()
   {
      _x++;
      _y++;
   }

   public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        DerivedClass o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine("x = {0}, y = {1}", o.X, o.Y);
    }
}
// Output: x = 111, y = 161
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

```
BaseClass bc = new BaseClass();   // Error
```

you will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

```
// compile with: /target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}
```

```
public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }

}
```

A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, DoWork on class F cannot call DoWork on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Methods

The sealed modifier is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class. A sealed class cannot also be an abstract class.

The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

In C# structs are implicitly sealed; therefore, they cannot be inherited.

```
using System;
sealed class MyClass
{
public int x;
public int y;
}

class MainClass
{
public static void Main()
{
MyClass mC = new MyClass();
mC.x = 110;
mC.y = 150;
Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
}
}
```

In the preceding example, if you attempt to inherit from the sealed class by using a statement like this:

```
class MyDerivedC: MyClass {} // Error
```

You will get the error message:
'MyDerivedC' cannot inherit from sealed class 'MyBaseC'.

In C# a method can't be declared as sealed. However when we override a method in a derived class, we can declare the overrided method as sealed as shown below. By declaring it as sealed, we can avoid further overriding of this method.

```
using System;
class MyClass1
{
public int x;
public int y;

public virtual void Method()
{
Console.WriteLine("virtual method");
}
}

class MyClass : MyClass1
{
public override sealed void Method()
{
Console.WriteLine("sealed method");
}
}

class MainClass
{
public static void Main()
{
MyClass1 mC = new MyClass();
mC.x = 110;
mC.y = 150;
Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
mC.Method();
}
}
```

Interfaces

An interface contains definitions for a group of related functionalities that a class or a struct can implement.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The following example shows an implementation of the IEquatable<T> interface. The implementing class, Car, must provide an implementation of the Equals method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        if (this.Make == car.Make &&
            this.Model == car.Model &&
            this.Year == car.Year)
        {
            return true;
        }
        else
            return false;
    }
}
```

If a class implements two interfaces that contain a member with the same signature, then implementing that member on the class will cause both interfaces to use that member as their implementation.

```
interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

```
class Test

{
    static void Main()
    {
        SampleClass sc = new SampleClass();
        IControl ctrl = (IControl)sc;
        ISurface srfc = (ISurface)sc;

       // all the calls to Paint invoke the same method.

        sc.Paint();
        ctrl.Paint();
        srfc.Paint();
    }
}
// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

It is possible to implement an interface member explicitly—creating a class member that is only called through the interface, and is specific to that interface. This is accomplished by naming the class member with the name of the interface and a period. For example:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

The class member IControl.Paint is only available through the IControl interface, and ISurface.Paint is only available through ISurface. Both method implementations are separate, and neither is available directly on the class. For example:

```
// Call the Paint methods from Main.


SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = (IControl)obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = (ISurface)obj;
```

```
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Explicit implementation is also used to resolve cases where two interfaces each declare different members of the same name such as a property and a method:

```
interface ILeft

{
    int P { get;}
}
interface IRight
{
    int P();
}
```

To implement both interfaces, a class has to use explicit implementation either for the property P, or the method P, or both, to avoid a compiler error. For example:

```
class Middle : ILeft, IRight

{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

The following example declares an interface, IDimensions, and a class, Box, which explicitly implements the interface members getLength and getWidth. The members are accessed through the interface instance dimensions.

```
interface IDimensions
{
    float getLength();
    float getWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.getLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.getWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = (IDimensions)box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.getLength());
        //System.Console.WriteLine("Width: {0}", box1.getWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.getLength());
        System.Console.WriteLine("Width: {0}", dimensions.getWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Notice that the following lines, in the Main method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a class instance:

```
System.Console.WriteLine("Length: {0}", box1.getLength());
System.Console.WriteLine("Width: {0}", box1.getWidth());
```

Notice also that the following lines, in the Main method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

```
System.Console.WriteLine("Length: {0}", dimensions.getLength());
System.Console.WriteLine("Width: {0}", dimensions.getWidth());
```

Icomparable and IComparer Interfaces

If you have an array of types (such as **string** or **integer**) that already support **IComparer**, you can sort that array without providing any explicit reference to **IComparer**. In that case, the elements of the array are cast to the default implementation of **IComparer** (**Comparer.Default**) for you. However, if you want to provide sorting or comparison capability for your custom objects, you must implement either or both of these interfaces.

The following .NET Framework Class Library namespace is referenced in this article:
System.Collections

IComparable

The role of **IComparable** is to provide a method of comparing two objects of a particular type. This is necessary if you want to provide any ordering capability for your object. Think of **IComparable** as providing a default sort order for your objects. For example, if you have an array of objects of your type, and you call the **Sort** method on that array, **IComparable** provides the comparison of objects during the sort. When you implement the **IComparable** interface, you must implement the **CompareTo** method, as follows:

```
// Implement IComparable CompareTo method - provide default sort order.
int IComparable.CompareTo(object obj)
{
   car c=(car)obj;
   return String.Compare(this.make,c.make);

}
```

The comparison in the method is different depending on the data type of the value that is being compared. **String.Compare** is used in this example because the property that is chosen for the comparison is a string.

IComparer

The role of **IComparer** is to provide additional comparison mechanisms. For example, you may want to provide ordering of your class on several fields or properties, ascending and descending order on the same field, or both.

Using **IComparer** is a two-step process. First, declare a class that implements **IComparer**, and then implement the **Compare** method:

```
private class sortYearAscendingHelper : IComparer
{
   int IComparer.Compare(object a, object b)
   {
      car c1=(car)a;
      car c2=(car)b;
      if (c1.year > c2.year)
         return 1;
      if (c1.year < c2.year)
         return -1;
      else
         return 0;
   }
}
```

Note that the **IComparer.Compare** method requires a tertiary comparison. 1, 0, or -1 is returned depending on whether one value is greater than, equal to, or less than the other. The sort order (ascending or descending) can be changed by switching the logical operators in this method.

The second step is to declare a method that returns an instance of your **IComparer** object:

```
public static IComparer sortYearAscending()
{
   return (IComparer) new sortYearAscendingHelper();
}
```

In this example, the object is used as the second argument when you call the overloaded **Array.Sort** method that accepts **IComparer**. The use of **IComparer** is not limited to arrays. It is accepted as an argument in a number of different collection and control classes.

Step-by-Step Example

The following example demonstrates the use of these interfaces. To demonstrate **IComparer** and **IComparable**, a class named **car** is created. The **car** object has the **make** and **year** properties. An ascending sort for the **make** field is enabled through the **IComparable** interface, and a descending sort on the **make** field is enabled through the **IComparer** interface. Both ascending and descending sorts are provided for the **year** property through the use of **IComparer**.

1. In Visual C#, create a new Console Application project. Name the application ConsoleEnum.
2. Rename Program.cs as Host.cs, and then replace the code with the following code.
   .

```
using System;

namespace ConsoleEnum
{
   class host
   {
      [STAThread]
      static void Main(string[] args)
      {
         // Create an arary of car objects.
         car[] arrayOfCars= new car[6]
         {
            new car("Ford",1992),
            new car("Fiat",1988),
            new car("Buick",1932),
            new car("Ford",1932),
            new car("Dodge",1999),
            new car("Honda",1977)
         };

         // Write out a header for the output.
         Console.WriteLine("Array - Unsorted\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo IComparable by sorting array with "default" sort order.
         Array.Sort(arrayOfCars);
         Console.WriteLine("\nArray - Sorted by Make (Ascending -
IComparable)\n");
```

```
         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo ascending sort of numeric value with IComparer.
         Array.Sort(arrayOfCars,car.sortYearAscending());
         Console.WriteLine("\nArray - Sorted by Year (Ascending - 
IComparer)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo descending sort of string value with IComparer.
         Array.Sort(arrayOfCars,car.sortMakeDescending());
         Console.WriteLine("\nArray - Sorted by Make (Descending - 
IComparer)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo descending sort of numeric value using IComparer.
         Array.Sort(arrayOfCars,car.sortYearDescending());
         Console.WriteLine("\nArray - Sorted by Year (Descending - 
IComparer)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         Console.ReadLine();
      }
   }
}
```

3. Add a class to the project. Name the class **car**.
4. Replace the code in Car.cs with the following:

```
using System;
using System.Collections;
namespace ConsoleEnum
{
   public class car : IComparable
   {
      // Beginning of nested classes.

      // Nested class to do ascending sort on year property.
      private class sortYearAscendingHelper: IComparer
      {
         int IComparer.Compare(object a, object b)
         {
            car c1=(car)a;
            car c2=(car)b;

            if (c1.year > c2.year)
               return 1;

            if (c1.year < c2.year)
         return -1;

      else
         return 0;
   }
}

// Nested class to do descending sort on year property.
private class sortYearDescendingHelper: IComparer
{
   int IComparer.Compare(object a, object b)
   {
      car c1=(car)a;
      car c2=(car)b;

      if (c1.year < c2.year)
         return 1;

      if (c1.year > c2.year)
         return -1;

      else
         return 0;
   }
}

// Nested class to do descending sort on make property.
private class sortMakeDescendingHelper: IComparer
{
   int IComparer.Compare(object a, object b)
   {
      car c1=(car)a;
      car c2=(car)b;
       return String.Compare(c2.make,c1.make);
   }
}

// End of nested classes.

private int year;
private string make;

public car(string Make,int Year)
{
   make=Make;
   year=Year;
}

public int Year
{
   get  {return year;}
   set {year=value;}
}

public string Make
```

```
      {
         get {return make;}
         set {make=value;}
      }

      // Implement IComparable CompareTo to provide default sort order.
      int IComparable.CompareTo(object obj)
      {
         car c=(car)obj;
         return String.Compare(this.make,c.make);
      }

      // Method to return IComparer object for sort helper.
      public static IComparer sortYearAscending()
      {
         return (IComparer) new sortYearAscendingHelper();
      }

      // Method to return IComparer object for sort helper.
      public static IComparer sortYearDescending()
      {
         return (IComparer) new sortYearDescendingHelper();
      }

      // Method to return IComparer object for sort helper.
      public static IComparer sortMakeDescending()
      {
        return (IComparer) new sortMakeDescendingHelper();
      }

   }
}
```

5. Run the project. The following output appears in the Console window:

```
Array - Unsorted

Ford            1992
Fiat            1988
Buick           1932
Ford            1932
Dodge           1999
Honda           1977

Array - Sorted by Make (Ascending - IComparable)

Buick           1932
Dodge           1999
Fiat            1988
Ford            1932
Ford            1992
Honda           1977

Array - Sorted by Year (Ascending - IComparer)
```

```
Ford            1932
Buick           1932
Honda           1977
Fiat            1988
Ford            1992
Dodge           1999

Array - Sorted by Make (Descending - IComparer)

Honda           1977
Ford            1932
Ford            1992
Fiat            1988
Dodge           1999
Buick           1932

Array - Sorted by Year (Descending - IComparer)

Dodge           1999
Ford            1992
Fiat            1988
Honda           1977
Buick           1932
Ford            1932
```

Polymorphism

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance.

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

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

```
class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        System.Collections.Generic.List<Shape> shapes = new
        System.Collections.Generic.List<Shape>();
        shapes.Add(new Rectangle());
        shapes.Add(new Triangle());
        shapes.Add(new Circle());

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (Shape s in shapes)
        {
            s.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

Null Types

The null keyword is a literal that represents a null reference, one that does not refer to any object. null is the default value of reference-type variables. Ordinary value types cannot be null.

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
```

As we all know, a value type variable cannot be null. That's why they are called Value Type. Value type has a lot of advantages, however, there are some scenarios where we require value type to hold null also.
Consider the following scenario:

Scenario 1: You are retrieving nullable integer column data from database table, and the value in database is null, there is no way you can assign this value to an C# int.

Scenario 2: Suppose you are binding the properties from UI but the corresponding UI don't have data. (for example model binding in Asp.Net MVC or WPF). Storing the default value in model for value type is not a viable option.

Scenario 3: In Java, java.Util.Date is a reference type, and therefore, the variable of this type can be set to null. However, in CLR, System.DateTime is a value type and a DateTime variable cannot be null. If an application written in Java wants to communicate a date/time to a Web service running on the CLR, there is a problem if the Java application sends null because the CLR has no way to represent this and operate on it.

Scenario 4: When passing value type parameter to a function, if the value of parameter is not know and if you don't want to pass it, you go with default value. But default value is not always a good option because default value can also a passed parameter value, so, should not be treated explicitly.

Scenario 5: When deserializing the data from xml or json, it becomes difficult to deal with the situation if the value type property expects a value and it is not present in the source.

To get rid of these situations, Microsoft added the concept of Nullable types to the CLR.

C# also supports simple syntax to use Nullable types. It also supports implicit conversion and casts on Nullable instances. The following example shows this:

```
// Implicit conversion from System.Int32 to Nullable<Int32>
int? i = 5;

// Implicit conversion from 'null' to Nullable<Int32>
int? j = null;

// Explicit conversion from Nullable<Int32> to non-nullable Int32
int k = (int)i;

// Casting between nullable primitive types
Double? x = 5; // Implicit conversion from int to Double? (x is 5.0 as a double)
Double? y = j; // Implicit conversion from int? to Double? (y is null)
```

You can use operators on `Nullable` types the same way you use it for the containing types.

- Unary operators (++, --, -, etc) returns `null` if the `Nullable` types value is set to `null`.

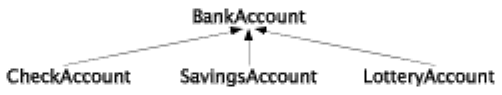
- Binary Operator (+, -, *, /, %, ^, etc) returns `null` if any of the operands is `null`.
- For Equality Operator, if both operands are `null`, expression is evaluated to `true`. If either operand is `null`, it is evaluated to `false`. If both are not `null`, it compares as usual.
- For Relational Operator (>, <, >=, <=), if either operand is `null`, the result is `false` and if none of the operands is `null`, compares the value.

See the example below:

```
int? i = 5;
int? j = null;

// Unary operators (+ ++ - -- ! ~)
i++; // i = 6
j = -j; // j = null

// Binary operators (+ - * / % & | ^ << >>)
i = i + 3; // i = 9
j = j * 3; // j = null;

// Equality operators (== !=)
if (i == null) { /* no */ } else { /* yes */ }
if (j == null) { /* yes */ } else { /* no */ }
if (i != j) { /* yes */ } else { /* no */ }

// Comparison operators (< > <= >=)
if (i > j) { /* no */ } else { /* yes */ }
```

Upcasting and downcasting

Upcasting converts an object of a specialized type to a more general type
Downcasting converts an object from a general type to a more specialized type



A specialization hierarchy of bank accounts

```
BankAccount    ba1,   ba2 =   new BankAccount("John", 250.0M, 0.01);

 LotteryAccount la1,  la2 =   new LotteryAccount("Bent", 100.0M);

   ba1 = la2;                   // upcasting   - OK

//  la1 = ba2;                  // downcasting - Illegal  -   discovered at compile time
```

// la1 = (LotteryAccount)ba2;   // downcasting – Illegal - discovered at run time

la1 = (LotteryAccount)ba1;   // downcasting - OK - ba1 already refers to a LotteryAccount

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an InvalidCastException to be thrown.

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
      //solution
      // if (a is Reptile)
      //    Reptile r = (Reptile)a;

    }

}
```

C# provides the is and as operators to enable you to test for compatibility before actually performing a cast.

As operator

You can use the as operator to perform certain types of conversions between compatible reference types or nullable types. The following code shows an example.

```
class csrefKeywordsOperators
```

```
{
    class Base
    {
        public override string  ToString()
        {
                    return "Base";
        }
    }
    class Derived : Base
    { }

    class Program
    {
        static void Main()
        {

            Derived d = new Derived();

            Base b = d as Base;
            if (b != null)
            {
                Console.WriteLine(b.ToString());
            }

        }
    }
}
```

The as operator is like a cast operation. However, if the conversion isn't possible, as returns null instead of raising an exception. Consider the following example:

```
expression as type
```

The code is equivalent to the following expression except that the expression variable is evaluated only one time.

```
expression is type ? (type)expression : (type)null
```

Note that the as operator performs only reference conversions, nullable conversions, and boxing conversions. The as operator can't perform other conversions, such as user-defined conversions, which should instead be performed by using cast expressions.

Is Operator

Checks if an object is compatible with a given type. For example, the following code can determine if an object is an instance of the MyObject type, or a type that derives from MyObject:

```
if (obj is MyObject)
{
}
```

An is expression evaluates to true if the provided expression is non-null, and the provided object can be cast to the provided type without causing an exception to be thrown.

The is keyword causes a compile-time warning if the expression is known to always be true or to always be false, but typically evaluates type compatibility at run time.

The is operator cannot be overloaded.

Note that the is operator only considers reference conversions, boxing conversions, and unboxing conversions.

Example

```
class Class1 {}
class Class2 {}
class Class3 : Class2 { }

class IsTest
{
    static void Test(object o)
    {
        Class1 a;
        Class2 b;

        if (o is Class1)
        {
            Console.WriteLine("o is Class1");
            a = (Class1)o;
            // Do something with "a."
        }
        else if (o is Class2)
        {
            Console.WriteLine("o is Class2");
            b = (Class2)o;
            // Do something with "b."
        }

        else
        {
            Console.WriteLine("o is neither Class1 nor Class2.");
        }
    }

     static void Main()
     {
        Class1 c1 = new Class1();
        Class2 c2 = new Class2();
        Class3 c3 = new Class3();
        Test(c1);
        Test(c2);
        Test(c3);
        Test("a string");
    }
}
```

```
/*
Output:
o is Class1
o is Class2
o is Class2
o is neither Class1 nor Class2.
*/
```

Nesting classes

A class can be created inside of another class. A class created inside of another is referred to as nested.
Here is an example of a class called Inside that is nested in a class called Outside:

```
public class Outside
{
    public class Inside
    {
    }
}
```

The name of a nested class is not "visible" outside of the nesting class. To access a nested class outside of the nesting class, you must qualify the name of the nested class anywhere you want to use it. For example, if you want to declare an Inside variable somewhere in the program but outside of Outside, you must qualify its name. Here is an example:

```
using System;

public class Outside
{
    public class Inside
    {
        public Inside()
        {
            Console.WriteLine(" -= Inside =-");
        }
    }

    public Outside()
    {
        Console.WriteLine(" =- Outside -=");
    }
}

public class Exercise
{
    static int Main()
    {
        Outside recto = new Outside();
        Outside.Inside ins = new Outside.Inside();

        return 0;
    }
}
```

This would produce:

```
=- Outside -=
 -= Inside =-
```

Because there is no programmatically privileged relationship between a nested class and its "container" class, if you want to access the nested class in the nesting class, you can use its static members. In other words, if you want, you can declare static all members of the nested class that you want to access in the nesting class. Here is an example:

```
using System;

public class Outside
    {
        public class Inside
        {
            public static string inMessage;

            public Inside()
            {
                Console.WriteLine(" -= Insider =-");
                inMessage = "Sitting inside while it's raining";
            }

            public static void Show()
            {
                Console.WriteLine("Show me the wonderful world of C# Programming");
            }
        }

        public Outside()
        {
            Console.WriteLine(" =- The Parent -=");
        }

        public void Display()
        {
            Console.WriteLine(Inside.inMessage);
            Inside.Show();
        }
    }

class Exercise
{
     static void Main(string[] args)
        {
            Outside recto = new Outside();
            Outside.Inside ins = new Outside.Inside();

            recto.Display();
           // return 0;

        }
}
```

This would produce:

=- The Parent -=

```
 -= Insider =-
Sitting inside while it's raining
Show me the wonderful world of C# Programming
```

In the same way, if you want to access the nesting class in the nested class, you can go through the static members of the nesting class. To do this, you can declare static all members of the nesting class that you want to access in the nested class. Here is an example:

```
using System;

public class Outside
{
        public class Inside
        {
                public static string InMessage;

                public Inside()
                {
                        Console.WriteLine(" -= Insider =-");
                        InMessage = "Sitting inside while it's raining";
                }

                public static void Show()
                {
                Console.WriteLine("Show me the wonderful world of C#
Programming");
                }

                public void FieldFromOutside()
                {
                        Console.WriteLine(Outside.OutMessage);
                }
        }

        private static string OutMessage;

        public Outside()
        {
                Console.WriteLine(" =- The Parent -=");
                OutMessage = "Standing outside! It's cold and raining!!";
        }

        public void Display()
        {
                Console.WriteLine(Inside.InMessage);
                Inside.Show();
        }
}

public class Exercise
{
    static int Main()
    {
        Outside recto = new Outside();
        Outside.Inside Ins = new Outside.Inside();

        Recto.Display();
        Console.WriteLine();
        Ins.FieldFromOutside();
        return 0;
    }
}
```

This would produce:

```
=- The Parent -=
 -= Insider =-
Sitting inside while it's raining
Show me the wonderful world of C# Programming

Standing outside! It's cold and raining!!
```

Instead of static members, if you want to access members of a nested class in the nesting class, you can first declare a variable of the nested class in the nesting class. In the same way, if you want to access members of a nesting class in the nested class, you can first declare a variable of the nesting class in the nested class. Here is an example:

```
using System;

public class Outside
{
        // A member of the nesting class
        private string OutMessage;

        // The nested class
        public class Inside
        {
                // A field in the nested class
                public string InMessage;

                // A constructor of the nested class
                public Inside()
                {
                        Console.WriteLine(" -= Insider =-");
                        this.InMessage = "Sitting inside while it's raining";
                }

                // A method of the nested class
                public void Show()
                {
                        // Declare a variable to access the nesting class
                        Outside outsider = new Outside();
                        Console.WriteLine(outsider.OutMessage);
                }
        } // End of the nested class

        // A constructor of the nesting class
        public Outside()
        {
                this.OutMessage = "Standing outside! It's cold and raining!!";
                Console.WriteLine(" =- The Parent -=");
        }

        // A method of the nesting class
        public void Display()
        {
                Console.WriteLine(insider.InMessage);
        }

        // Declare a variable to access the nested class
        Inside insider = new Inside();
}

public class Exercise
{
    static int Main()
    {
        Outside recto = new Outside();
        Outside.Inside Ins = new Outside.Inside();

        Ins.Show();
        Recto.Display();
        return 0;
    }
}
```

This would produce:

```
-= Insider =-
 =- The Parent -=
 -= Insider =-
 -= Insider =-
 =- The Parent -=
Standing outside! It's cold and raining!!
Sitting inside while it's raining
```

Extension methods

Extension methods are used to "extend" functionality of an existing class. Extension method is defined separately from the class it is extending. Here are the main rules that define Extension Methods:

Extension methods can be defined on classes, structures and interfaces

The method is defined separately from the class it is extending. It should be defined as static and be in non-generic static class that is not nested.

Extension method should have an “instance parameter” defined. This is the first parameter in Extension method and that is preceded by this modifier. The instance parameter cannot be a pointer type and cannot have parameter modifiers (ref, out, etc..)

There is no restriction on the name of extension method. This means that you may have several extension method with the same name and even with the same name as of some instance method. Method to call is selected in the following order:

1. Instance method
2. Extension method within the same namespace
3. Extension method outside the current namespace

Example

Lets assume you have a class **Cat:**

```
using System;

namespace  ExtensionMethodsSamples
{
    public class Cat
    {
        public void Eat(object food)
        {
            //TODO: eat implementation
        }

        public void Jump()
        {
            //TODO: jump implementation
        }

    }
}
```

After some time you want to extend it with **Dance** method. To create is as extension method you declare it in a separate class as a static method in a following way:

```
using System;
using ExtensionMethodsSamples;

namespace CatExtension
{
    public static class CatExtension
    {
        public static void Dance(this Cat cat)
        {
            for (int i=0; i<10; i++)
                cat.Jump(); // dumb implementation, forgive me, cat
        }
    }
}
```

Note **this** keyword in parameter declaration. Also note that method is static and that it is defined in namespace **CatExtension**(name doesn't matter, just note that it is different from the one where **Cat** is defined).

Now you should import namespace where extension method is defined and cat gains ability to "dance". You can call now **Dance** method of the **Cat** class just like the usual instance method.

```
using System;
using CatExtension;

namespace ExtensionMethodsSamples
{
    class Program
    {
        static void Main(string[] args)
        {
            Cat c = new Cat();
            c.Dance();

        }
    }
}
```

So, when and how to use Extension Methods?

The possible right uses of extension methods:

- To provide extension to a library that cannot be extended in other way and on which you do not have access over.

To use with Interfaces. Interfaces cannot include behavior by default. But now you can add it with Extensio.

Oveloading operators

This example shows how you can use operator overloading to create a complex number class Complex that defines complex addition. The program displays the imaginary and the real parts of the numbers and the addition result using an override of the ToString method.

Example

```
public struct Complex
{
    public int real;
    public int imaginary;

    // Constructor.
    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Specify which operator to overload (+),
    // the types that can be added (two Complex objects),
    // and the return type (Complex).
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
    }

    // Override the ToString() method to display a complex number
    // in the traditional format:
    public override string ToString()
    {
        return (System.String.Format("{0} + {1}i", real, imaginary));
    }
}

class TestComplex
{
    static void Main()
    {
        Complex num1 = new Complex(2, 3);
        Complex num2 = new Complex(3, 4);

        // Add two Complex objects by using the overloaded + operator.
        Complex sum = num1 + num2;

        // Print the numbers and the sum by using the overridden
        // ToString method.
        System.Console.WriteLine("First complex number:  {0}", num1);
        System.Console.WriteLine("Second complex number: {0}", num2);
        System.Console.WriteLine("The sum of the two numbers: {0}", sum);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    First complex number:  2 + 3i
    Second complex number: 3 + 4i
    The sum of the two numbers: 5 + 7i
*/
```

Class versus struct

Syntactical Comparison:

Now, semantics aside, there are a lot of things that are similar between **struct** and **class** in C#, but there are also a fair number of surprising differences. Let's look at a table that sums them up:

Feature	***Struct***	***Class***	***Notes***
Is a reference type?	No	Yes*	
Is a value type?	Yes	No	
Can have nested Types (enum, class, struct)?	Yes	Yes	
Can have constants?	Yes	Yes	
Can have fields?	Yes*	Yes	*Struct instance fields cannot be initialized, will automatically initialize to default value.*
Can have properties?	Yes	Yes	
Can have indexers	Yes	Yes	
Can have methods?	Yes	Yes	
Can have events?	Yes*	Yes	*Structs, like classes, can have events, but care must be taken that you don't subscribe to a* **copy** *of a struct instead of the struct you intended.*
Can have static members (constructors, fields, methods, properties, etc.)?	Yes	Yes	
Can inherit?	No*	Yes*	*Classes can inherit from other classes*

			(or object by default). Structs always inherit from System.ValueType and are sealed implicitly
Can implement interfaces?	Yes	Yes	
Can overload constructor?	Yes*	Yes	*Struct overload of constructor does not hide default constructor.*
Can define default constructor?	No	Yes	*The struct default constructor initializes all instance fields to default values and cannot be changed.*
Can overload operators?	Yes	Yes	
Can be generic?	Yes	Yes	
Can be partial?	Yes	Yes	
Can be sealed?	Always*	Yes	*Structs are always sealed and can never be inherited from.*
Can be referenced in instance members using *this* keyword?	Yes*	Yes	*In structs, this is a value variable, in classes, it is a readonly reference.*
Needs *new* operator to create instance?	No*	Yes	*C# classes must be instantiated using new. However, structs do not require this. While new can be used on a struct to call a constructor, you can elect not to use new and init the fields yourself, but you must init all fields and the fields must be public!*

Many of the items in this table should be fairly self explanatory. As you can see there the majority of behavior is identical for both **class** and **struct** types, however, there are several differences which can become potential pitfalls if not respected and understood!

Value vs. Reference Type

The fact that a **struct** is a *value* type, while **class** is a *reference* type. This has several ramifications:

With value types:

- **Value type assignments copy all members the whole value** - *this copies all members of one value to another making two complete instances.*
- **Value types passed by parameter or returned from methods/properties copy whole value** - *this behavior is the same as value assignment.*
- **Value types assigned to an *object* are *boxed*** *– that is, they are surrounded by an object and then passed by reference.*
- **Value types are destroyed when they pass out of scope -** *local variables and parameters are typically cleaned up when scope is exited, members of an enclosing type are cleaned up when the enclosing type is cleaned up.*
- **Value types may be created on the stack or heap as appropriate** - *typically parameters and locals are created on the stack, members of an enclosing class are typically on the heap.*
- **Value types cannot be null** - *default value of members that are primitive is zero, default value of members that are a struct is an instance with all struct members defaulted.*

In contrast, with reference types:

- **Reference type assignment only copies the reference** - *this decrements/increments reference counts as appropriate.*
- **Reference types passed by parameter or returned from methods/properties pass a reference** - *the reference is copied, but both references refer to the same original object.*
- **Reference types are only destroyed when garbage collected** - *after all references to the object are determined to be unreachable.*
- **Reference types are generally on the heap** - *it's always possible compiler may optimize, but in general you should always think of them as heap objects*
- **Reference types can be null -** *default value of members that are reference types members is null.*

```
public class PointClass
{
      public int X { get; set; }
     public int Y { get; set; }

}

 public struct PointStruct
{
    public int X { get; set; }
   public int Y { get; set; }

}

public static class Program
{
    public static void Main()
   { // assignment from one reference to another simply copies reference and increments
      //reference count.

       // In this case both references refer to one single object

        var pointClass = new PointClass { X = 5, Y = 10 };

        var copyOfPointClass = pointClass;

        // modifying one reference modifies all references referring to the same object

        copyOfPointClass.X = 0;

        // this will output [0, 10] even though it's our original reference because they are both
        //referring to the same object

       Console.WriteLine("Original pointClass is [{0},{1}]", pointClass.X, pointClass.Y);

        // ...
        // assignment from one struct to another makes a complete copy, so there are two separate
        //PointStruct each  with their own X and Y


        var pointStruct = new PointStruct { X = 5, Y = 10 };

        var copyOfPointStruct = pointStruct;
        // modifying one reference modifies all references referring to the same object

        copyOfPointStruct.X = 0;

        // the output will be [5,10] because the original pointStruct is unaffected by changes to the
        //copyOfPOintStruct

        Console.WriteLine("Original pointStruct is [{0},{1}]", pointStruct.X, pointStruct.Y);

      }
  }
```

Namespace

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code. Implementing Namespaces in your own code is a good habit because it is likely to save you from problems later when you want to reuse some of your code. For example, if you created a class named Console, you would need to put it in your own namespace to ensure that there wasn't any confusion about when the System.Console class should be used or when your class should be used. Generally, it would be a bad idea to create a class named Console, but in many cases your classes will be named the same as classes in either the .NET Framework Class Library or a third party library and namespaces help you avoid the problems that identical class names would cause.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface SampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace SampleNamespace.Nested
    {
        class SampleClass2 { }
    }
}
```

The following example shows how to call a static method in a nested namespace.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

Using Directive

```
// Namespace Declaration
using System;
using csharp_station.tutorial;

// Program start class
class UsingDirective
{
   // Main begins program execution.
   public static void Main()
   {
      // Call namespace member
      myExample.myPrint();
   }
}

// C# Station Tutorial Namespace
namespace csharp_station.tutorial
{
   class myExample
   {
      public static void myPrint()
      {
         Console.WriteLine("Example of using a using directive.");
      }
   }
}
```