Classes

Class – prototype, that defines data and the methods that work on that data.

1. Defining a Class

[*attributes*] [*modifiers*] class <*class_name*> [:<*base_class_name*>]
{
    // class body
}[;]

2. Class Members

- **field (member variable) – holds a value; modifiers: static, readonly, and const**
- **method (member function) – code that acts on the object's data (field values)**
- **property (smart field) – method that looks like a field to the class's clients**
- **constant – field with a value that can't be changed**
- **indexer (smart array) – member that enables to work with a class that's logically an array of data, as though the class itself were an aray**
- **event – causes some peace of code to run when something is happen**
- **operator – standard mathematical operator to a class via operator overloading**

3. Access Modifiers

public	**The member is accessible from outside the class's definition and hierarchy of derived classes.**
protected	**The member isn't visible outside the class and can be accessed by derived classes only.**
private	**The member can't be accessed outside the scope of the defining class and its derived classes (by default).**
internal	**The member is visible only within the current compilation unit.**

4. Method Main – the application's entry point; must be defined as static.

- **command-line arguments – a string array type as its only arguments**
- **return value – terminates the execution of the method**
  - **usually doesn't return a value – void;**
  - **a value of type int – shows an error level to the calling application to indicate user-defined success or failure (for console applications)**
- **multiple Main methods**

  /main:<*class_name*> **switch with the C# compiler specifies which class's Main method to use.**

5. Constructors

- **are called when ever an instance of the class is created with new**
- **have the same names as a class name**
- **initialize objects**
- **don't return values**

<*class*> <*object*> = new <*class*> (*constructor arguments*)

new **creates a new instance of a class:**

- **on the heap – reference types**
- **on the stack – value types**

6. Static Members and Instance Members

6.1. Instance Member – a copy of that member is made for every instance of the class (by default)

6.2. Static Member (static)

- **only one copy of the member exists**
- **a static member is created when the application containing the class is loaded**
- **exists throughout the life of the application**
- **the member is accessible even before the class has been instantiated**

7. Constructor Initializers

All constructors first invoke the base class's constructor. The constructor initializers specify which class and which constructor is called.

base (…) – **calls the current class's base class constructor**

this (…) – **calls another constructor defined within the class itself**

8. Constant

- **field that remain constant for the life of the application**
- **defined with the const keyword**
- **the constant value is set at compile time**
- **by default is a static**

9. Read-Only Fields

readonly – **constant field that value is set in the constructor at run time**

static readonly – **constant field that value is set in the static constructor that by default is public**

Example: Class Point – instance members

```
using System;
class Point
{
  private int x, y;                    // Coordinates of a point
  // Default constructor without parameters
  public Point ()
  {
     x = 0;
     y = 0;
  }
  // Constructor with two parameters
  public Point (int initialX, int initialY)
  {  x = initialX;
     y = initialY;
  }
  // Instance method – calculates the distance to the point other
  public double DistanceTo (Point other)
  {
     int xDiff = x - other.x;
     int yDiff = y - other.y;
     return Math.Sqrt (xDiff * xDiff + yDiff * yDiff);
  }
  // Overriding method System.Object.ToString
  public override string ToString()
  {
     return "("+x+", "+y+")";
  }
}
```

```
class PointApp
{
  static void Main (string[] args)
  {
     Point origin = new Point ();
     Point bottomRight = new Point (600,800);
     double distance = bottomRight.DistanceTo (origin);
     Console.WriteLine
           ("The distance between the points {0} and {1} is {2}.",
            bottomRight, origin, distance);
  }
}
```

Results:

The distance between the points (600, 800) and (0, 0) is 1000.

Example: Class Point – static fields

```
using System;
class Point
{
  private int x, y;
  private static int objectCount = 0;
  public Point ()
  {
     x = 0;
     y = 0;
     objectCount++;
  }
  public Point (int initialX, int initialY)
  {
     x = initialX;
     y = initialY;
     objectCount++;
  }
```

```
   public double DistanceTo (Point other)
   {
      int xDiff = x - other.x;
      int yDiff = y - other.y;
      return Math.Sqrt (xDiff * xDiff + yDiff * yDiff);
   }
   public override string ToString ()
   {
      return "("+x+", "+y+")";
   }
   public static int ObjectCount ()
   {
      return objectCount;
   }
}
```

```
class PointApp
{
   static void Main (string[] args)
   {
      Console.WriteLine ("The number of Point objects: {0}",
                          Point.ObjectCount ());
      Point origin = new Point ();
      Point bottomRight = new Point (600, 800);
      double distance = bottomRight.DistanceTo (origin);
      Console.WriteLine
                ("The distance between {0} and {1} is {2}.",
                 bottomRight, origin, distance);
      Console.WriteLine("The number of Point objects: {0}",
                          Point.ObjectCount ());
   }
}
```

Results:

The number of Point objects: 0
The distance between (600, 800) and (0, 0) is 1000.
The number of Point objects: 2

Example:

Application that keeps a track of the current workstation's IP address (the workstation obtains its IP address dynamically):

- **with** readonly **field**
- **with** static readonly **field**

Class System.Net.IPAddress – **provides an Internet Protocol (IP) address.**

Class System.Net.Dns – **provides simple domain name system (DNS) resolution functionality.**

Method Dns.Resolve – **queries a DNS server for the IP address associated with a host name or IP address.**

public static IPHostEntry Resolve (string hostName);

IPHostEntry **provides a container class for Internet host address information. When** hostName **is a DNS-style host name associated with multiple IP addresses, only the first IP address that resolves to that host name is returned.**

Property IPHostEntry.AddressList – **gets/sets a list of IP addresses that are associated with a host.**

```
using System;
using System.Net;
class Workstation
{  public const string HostName = "FKSU2300A-3";
   public readonly string IPAddressString;
   public Workstation ()
   {  IPAddress ipAddress =
                        Dns.Resolve(HostName).AddressList[0];
      IPAddressString = ipAddress.ToString();
   }
}
class GetIpAddress
{  static void Main (string[] args)
   {  Workstation workstation = new Workstation();
      Console.WriteLine ("The IP address of '{0}' is {1}.",
         Workstation.HostName, workstation.IPAddressString);
   }
}
```

Results:

The IP address of 'FKSU2300A-3' is 81.161.244.49.

```
using System;
using System.Net;
class Workstation
{   public const string HostName = "FKSU2300A-3";
    public static readonly string IPAddressString;
    static Workstation ()
    {   IPAddress ipAddress =
                          Dns.Resolve(HostName).AddressList[0];
        IPAddressString = ipAddress.ToString();
    }
}
class GetIpAddress
{   static void Main (string[] args)
    {   Console.WriteLine ("The IP address of '{0}' is {1}.",
            Workstation.HostName, Workstation.IPAddressString);
    }
}
```

Results:

The IP address of 'FKSU2300A-3' is 81.161.244.49.

Inheritance

Inheritance – a class is built upon another class, in terms of data or behaviour.

class *<derived_class>* : *<base_class>*

- public, protected **or** internal **members can be inherited**
- **constructor can not be inherited – each subclass has to implement its constructor**
- **C# does not support multiple inheritance through derivation, but implements multiple interfaces**
- **sealed classes** (sealed) **– can never have any derived classes**

Example: Class Point3D **inherits the class** Point

```
using System;
class Point
{
  private int x,y;
  public Point ()
  {   x = 0;
      y = 0;
  }
  public Point (int initialX, int initialY)
  {   x = initialX;
      y = initialY;
  }
  public void Move (int dx, int dy)
  {   x+=dx;
      y+=dy;
  }
  public override string ToString ()
  {
     return x+","+y;
  }
}
class Point3D : Point
{   private int z;
    public Point3D () : base ()
    {
       z = 0;
    }
    public Point3D (int initialX, int initialY, int initialZ) :
                                          base (initialX, initialY)
  {
       z = initialZ;
  }
    public void Move (int dx, int dy, int dz)
    {   base.Move (dx, dy);
        z += dz;
    }
    public override string ToString ()
    {
       return base.ToString ()+","+z;
    }
}
class PointApp
{   static void Main (string[] args)
    {   Point3D point = new Point3D (600,800,1000);
        point.Move (1, 1, 1);
        Console.WriteLine ("({0})", point);
    }
}
```

Results:

(601,801,1001)

Structs

Struct

- **value type**
- **contains different data type**
- **is referred to as a lightweight version of a class**

1. Defining a Struct

[*attributes*] [*modifiers*] struct *<struct_name>* [: *<interfaces>*]

```
{
    // struct body
}[;]
```

2. Struct Members

- **constructors**
- **constants**
- **fields**
- **methods**
- **properties**
- **indexers**
- **operators**
- **nested types**

3. Limitations

- **no default constructor**
- **defined as sealed – it can't serve as a base class**
- **implicitly derived from** System.ValueType **– the ultimate base class of all value types**

4. Usage

- **contain very small data**
- **contain few or even no methods to access or modify the contained data**

Example: Define the standard RGB **struct with static fields to hold the red, green, and blue values.**

```
using System;
struct RGB
{   public static readonly RGB RED     = new RGB (255,0,0);
    public static readonly RGB GREEN= new RGB (0,255,0);
    public static readonly RGB BLUE   = new RGB (0,0,255);
    public static readonly RGB WHITE = new RGB (255,255,255);
    public static readonly RGB BLACK = new RGB (0,0,0);
    public int Red;
    public int Green;
    public int Blue;
    public RGB (int red, int green, int blue)
    {
          Red    = red;
          Green = green;
          Blue    = blue;
    }
```

```
    public override string ToString ()
    {   return Red.ToString("X2") + Green.ToString("X2") +
                Blue.ToString("X2");
    }
}
class StructApp
{   static void PrintRGBValue (string color, RGB rgb)
    {
        Console.WriteLine ("The value for {0} is {1}", color, rgb);
    }
    static void Main(string[] args)
    {   PrintRGBValue ("red  ",   RGB.RED);
        PrintRGBValue ("green", RGB.GREEN);
        PrintRGBValue ("blue ",  RGB.BLUE);
        PrintRGBValue ("white",  RGB.WHITE);
        PrintRGBValue ("black",  RGB.BLACK);
    }
}
```

Results:

```
The value for red     is FF0000
The value for green is 00FF00
The value for blue    is 0000FF
The value for white  is FFFFFF
The value for black  is 000000
```

Methods

Methods (member functions) – give classes their behavioral characteristics.

Method Parameters

Value and Reference Parameters

1. **Value-type method parameters – passing by value**

- **a copy of the value is passed to the method**
- **if the called method makes changes to the data through the value-type incoming parameters the changes don't affect the variables passed down from the calling method**

2. Reference-type method parameters – passing by reference

- **a copy of the reference (another reference to the same data) is passed to the method**
- **if the called method makes changes to the data through the reference, the changes are made to the original data and the changes will be available to the calling method when the called method returns**

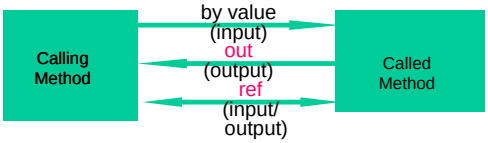
3. Returning More than a Single Value

3.1 ref Method Parameters

- **ref parameters point to the same memory as the variables in the calling code**
- **if the called method modifies the values the calling code's variables are modified (pointers in C++)**
- **limitation – ref parameters have to be initialized before calling the method**

3.2 out Method Parameters

- **out parameters don't require the calling code to initialized the passed arguments first**
- **must be modified in the called method**



Example: Passing value-type method parameters

```
using System;
class SomeClass
{   public void Change(int x, ref int y, out int z)
    {   x += 5;                  // input parameter
        y *= x;                  // input/output parameter
        z = 10 * y;              // output parameter
    }
}
class Test
{   static void Main()
    {   SomeClass sc = new SomeClass ();
        int v1 = 5, v2 = 5, v3;
        sc.Change (v1, ref v2, out v3);
        Console.WriteLine ("v1={0}, v2={1}, v3={2}", v1, v2, v3);
    }
}
```

Results:

v1=5, v2=50, v3=500

Пример: Passing reference-type method parameter

```
using System;
class AnotherClass
{
  public int ID;
}
class SomeClass
{
  public void ChangeObject (AnotherClass x,
                            ref AnotherClass y, out AnotherClass z)
  {   x.ID += 5;
      y.ID *= x.ID;
      z = new AnotherClass();
      z.ID = 10 * y.ID;
  }
}
```

```
class Test
{
   static void Main()
   {
      SomeClass sc    = new SomeClass ();
      AnotherClass r1 = new AnotherClass ();
      r1.ID = 5;
      AnotherClass r2 = new AnotherClass ();
      r2.ID = 5;
      AnotherClass r3;
      sc.ChangeObject (r1, ref r2, out r3);
      Console.WriteLine("r1.ID={0}, r2.ID={1}, r3.ID={2}",
                        r1.ID, r2.ID, r3.ID);
   }
}
```

Results:

r1.ID=10, r2.ID=50, r3.ID=500

```
class SomeClass
{
  public void ChangeObject (AnotherClass x, ref AnotherClass y,
                            out AnotherClass z)
  {
     x = new AnotherClass();
     x.ID += 5;
     y.ID *= x.ID;
     z = new AnotherClass();
     z.ID = 10 * y.ID;
  }
}
```

Results

r1.ID=5, r2.ID=25, r3.ID=250

Method Overloading

Method overloading – the same method name can be used multiple time with only the passed arguments changed.

1. **The behavior of the method differs slightly depending on the value types passed.**
2. **Method's parameter list must be different.**
3. **The method's return type and access modifier must be the same.**

Overloading Constructors

```
class Point
{
  private int x = 0;
  private int y = 0;
  // The constructor Point (int x, int y) is implicitly compiled
  // as though it were written public Point (int x, int y) : this ().
  // this () resolves to a call to public Point ().
  public Point (int x, int y)
  {  this.x = x;
     this.y = y;
  }
  public Point ()
  {  x = 0;
     y = 0;
  }
}
```

Constructor Initializer list – explicitly call one constructor from another:

```
public Point () : this (0, 0)
{
   ...
}
```

Inheritance and Overloading

The derived class can use overloaded method (as in Java).

Variable Method Parameters

The number of method arguments are known at run time.

The variable number of method parameters is specified by using the params keyword and by specifying an array in the method's argument list.

Example:

```
using System;
class Point
{   public int x;
    public int y;
    public Point (int x, int y)
    {   this.x=x;
        this.y=y;
    }
}
class Polygon
{   public void DrawPolygon (params Point[] p)
    {   Console.WriteLine("Polygon with  vertexes: ");
        for (int i=0; i<p.GetLength(0); i++)
            Console.WriteLine ("{0},{1}", p[i].x, p[i].y);
    }
}
```

```
class TestPolygon
{   static void Main()
    {   Point p1    = new Point(5,10);
        Point p2    = new Point(10,15);
        Point p3    = new Point(5,20);
        Polygon p = new Polygon();
        p.DrawPolygon (p1, p2, p3);
    }
}
```

or

```
static void Main()
{   Point[] pts = {new Point(5,10), new Point(10,15),
                   new Point(5,20)};
    Polygon p  = new Polygon();
    p.DrawPolygon (pts);
}
```

Example: Parameters of any type – params **parameter of type** object**.**

```
using System;
class Point
{   public int x;
    public int y;
    public Point (int x, int y)
    {   this.x = x;
        this.y = y;
    }
}
class OpenEnded
{   public void Foo (params object[] p)
    {   for (int i=0; i<p.GetLength(0); i++)
            Console.WriteLine (p[i]);
    }
}
```

```
class TestOpenEnded
{
    static void Main()
    {
        OpenEnded oe = new OpenEnded();
        oe.Foo(123,456, "Hello", new Point(7,8),9.0m,true,'X');
    }
}
```

Results:

```
123
456
Hello
Point
9,0
True
X
```

Virtual Methods

Virtual methods – modify the behavior of the base class in the derived class.

1. **Method Overriding – using** new **keyword with the derived class's method definition – hides the base class method (**new **is implicit default).**

Example: Method overriding using new

```
using System;
class Point
{
  protected int x = 0;
  protected int y = 0;
  public Point (int x, int y)
  {   this.x = x;
      this.y = y;
  }
```

```
  public void Move (int dx, int dy)
  {   x += dx;
      y += dy;
  }
  override public string ToString()
  {   return "("+x+","+y+")";
  }
}
class SlowPoint : Point
{
  private int xLimit;          // -xLimit <= x <= xLimit
  private int yLimit;          // -yLimit <= y <= yLimit
  public SlowPoint (int x, int y, int lower, int upper) : base (x, y)
  {
      xLimit = lower;
      yLimit = upper;
  }
```

```
   new public void Move (int dx, int dy)
   {
      x += dx;
      y += dy;
      x = Limit (x, xLimit);
      y = Limit (y, yLimit);
   }

   private int Limit (int d, int l)
   {
       // If the coordinate of the point d is greater than the upper
       // limit l, it becomes equal to the upper limit; if it is less than
       // the lower limit –l than it becomes equal to the lower limit
       // –l.
       return d > l ? l : d < -l ? -l : d;
   }
}
```

```
class TestOverriding
{
    static void Main (string[] args)
    {
       Point p1 = new Point (10, 20);
       p1.Move (5, 5);
       Console.WriteLine (p1);
       SlowPoint p2 = new SlowPoint (100, 200, 500, 800);
       p2.Move (600, 100);
       Console.WriteLine (p2);
    }
}
```

Results:
(15,25)
(500,300)

2. Polymorphism (many forms) – define a method multiple times throughout the class hierarchy so that the runtime calls the appropriate version of the method for the specific object being used

- **override a method in the class hierarchy using:**
  - virtual **keyword in the base class and**
  - override **keyword in the derived class**
- **declare an object from the base class**
- **runtime calls the appropriate method for the specific object**

Example: Method overriding without polymorphism – using new (early binding)

```
using System;
class Employee                          // Employee
{
   private string name;                 // Name
   private string address;              // Address
   private string phone;                // Telephone number
   private double payRate;              // Pay rate
```

```
   public Employee (string name, string address,
                    string phone, double payRate)
   {  this.name    = name;
      this.address = address;
      this.phone   = phone;
      this.payRate = payRate;
   }
   public double pay()
   {
      return payRate;
   }
   override public string ToString()
   {
      return name + ", " + address + ", " + phone;
   }
}
```

```
class Executive : Employee
{
    private double bonus;
    public Executive (string name, string address, string phone,
                      double payRate, double bonus) :
                      base(name, address, phone, payRate)
    {
        this.bonus=bonus;
    }
    new public double pay()          // Override pay
    {
        return base.pay() + bonus;
    }
}
```

```
class Hourly : Employee
{
    private int hoursWorked;
    public Hourly (string name, string address, string phone,
                   double payRate, int hoursWorked) :
                   base (name, address, phone, payRate)
    {
        this.hoursWorked = hoursWorked;
    }
    new public double pay()          // Override pay
    {
        return base.pay() * hoursWorked;
    }
}
```

```
class TestNotPolymorphic
{
    static void Main()
    {
        Employee e;
        e = new Executive("John","Sofia","1234567",400,100);
        Console.WriteLine(e + " " + e.pay());
        e = new Hourly("Maria", "Plovdiv", "765432", 20, 10);
        Console.WriteLine(e + " " + e.pay());
    }
}
```

Results:

John, Sofia, 1234567 400
Maria, Plovdiv, 765432 20

Example: Method overriding with polymorphism – using virtual and override (late binding)

```
using System;
class Employee
{
   private string name;
   private string address;
   private string phone;
   private double payRate;
   public Employee (string name, string address,
                    string phone, double payRate)
   {  this.name    = name;
      this.address = address;
      this.phone   = phone;
      this.payRate = payRate;
   }
   virtual public double pay()       // Virtual pay method
   {
      return payRate;
   }
   override public string ToString()
   {
      return name + ", " + address + ", " + phone;
   }
}
class Executive : Employee
{  private double bonus;
   public Executive (string name, string address, string phone,
                     double payRate, double bonus) :
                     base(name, address, phone, payRate)
   {
      this.bonus = bonus;
   }
   override public double pay()     // Override the virtual pay
   {                                // method
      return base.pay() + bonus;
   }
}
class Hourly : Employee
{  private int hoursWorked;
   public Hourly (string name, string address, string phone,
                  double payRate, int hoursWorked) :
                  base (name, address, phone, payRate)
   {
      this.hoursWorked = hoursWorked;
   }
   override public double pay()     // Override the virtual pay
   {                                // method
      return base.pay()*hoursWorked;
   }
}
```

```
class TestPolymorphic
{
   static void Main()
   {
      Employee e;
      e = new Executive ("John","Sofia","1234567",400,100);
      Console.WriteLine (e + " " + e.pay());
      e = new Hourly ("Maria","Plovdiv","765432",20,10);
      Console.WriteLine (e + " " + e.pay());
   }
}
```

Results:

John, Sofia, 1234567 500
Maria, Plovdiv, 765432 200

Rules:

1. override **method and** virtual **method must have the same access modifier (**protected**,** public**,** internal**)**
2. virtual **member can not be declared as** private **(can not be overridden); can be declared as** protected **(but not be used out of the class hierarchy).**

Example: Overriding the class Object methods

```
using System;
class Point                          // Inherits Object by default
{
  private int x, y;
  public Point ()
  {
    x = 0;
    y = 0;
  }
  public Point (int initialX, int initialY)
  {
    x = initialX;
    y = initialY;
  }
```

```
  // Overrides Object.Equals
  public override bool Equals (Object obj)
  {
    // Checks for null and compares the types at run time
    if (obj == null || GetType () != obj.GetType ())
        return false;
    Point p = (Point)obj;
    return (x == p.x) && (y == p.y);
  }
  // Overrides Object.GetHashCode
  public override int GetHashCode ()
  {
    // Generates a hash code using an XOR (exclusive OR)
    // operation
    return x ^ y;
  }
```

```
  // Overrides Object.ToString
  public override string ToString ()
  {
    return x + "," + y;
  }
}

class Point3D : Point                // Inherits Point
{
  private int z;
  public Point3D () : base ()
  {
    z = 0;
  }
```

```
  public Point3D (int initialX, int initialY, int initialZ) :
                    base (initialX, initialY)
  {    z = initialZ;   }
  // Overrides Point.Equals
  public override bool Equals (Object obj)
  {
    return base.Equals (obj) && z == ((Point3D)obj).z;
  }
  // Overrides Point.GetHashCode
  public override int GetHashCode()
  {
    return base.GetHashCode () ^ z;
  }
  // Overrides Point.ToString
  public override string ToString ()
  {   return base.ToString () + "," + z;   }
}
```

```
class InheritancePointApp
{
  static void Main (string[] args)
  {  Point3D point1 = new Point3D (100, 100, 100);
     Console.WriteLine ("The hash code of the point ({0}): {1}",
                        point1, point1.GetHashCode());
     Point3D point2 = new Point3D (10, 10, 10);
     Console.WriteLine ("The hash code of the point ({0}): {1}",
                        point2, point2.GetHashCode());
     Point3D point3 = new Point3D (10, 10, 10);
     Console.WriteLine ("The hash code of the point ({0}): {1}",
                        point3, point3.GetHashCode());
     Console.WriteLine ("({0}) and ({1}) are {2}.", point1, point2,
           point1.Equals (point2) ? "equal" : "not equal");
     Console.WriteLine ("({0}) and ({1}) are {2}.", point2, point3,
           point2.Equals (point3) ? "equal" : "not equal");
  }
}
```

Results:

```
The hash code of the point (100,100,100): 100
The hash code of the point (10,10,10): 10
The hash code of the point (10,10,10): 10
(100,100,100) and (10,10,10) are not equal.
(10,10,10) and (10,10,10) are equal.
```

Static Methods

Static method

- **exists in a class as a whole, rather than in a specific instance of the class**
- **defines using the static keyword**
- **cannot be referenced through an instance; instead, it is referenced through the type name:**

  class.method

1. Access to Class members

- **can access any static member within the class**
- **can't access an instance member**

2. Static Constructor

- **a class can have only one static constructor**
- **can't take parameters**
- **can't access instance members (including the this pointer)**
- **is executed before the first instance of a class is created**
- public **and** private **modifiers are not allowed**
- **can provide a nonstatic constructor with the same signature as the static constructor (the static constructor is called first)**
- **is executed before any static member (either data or function) of the class is accessed**

Example:

```
using System;
class Point
{
   private int x;
   private int y;
   private static int count;
   static Point()                    // Static constructor
   {
      count = 0;
   }
   public Point()                    // Nonstatic constructor
   {
       x = 0;
       y = 0;
       count++;
   }
   public Point (int x, int y)       // Constructor with two parameters
   {    this.x = x;
        this.y =y;
        count++;
   }
   public void Move (int dx, int dy)
   {    x += dx;
        y += dy;
   }
   override public string ToString()
   {
      return "(" + x + "," + y +")";
   }
   public static void Info()                    // Static method
   {
      Console.WriteLine("The number of points is " + count);
   }
}
```

```
class TestStaticMethod
{  static void Main()
   {  Point.Info ();
      Point p1 = new Point ();
      p1.Move(5,5);
      Console.WriteLine (p1);
      Point p2 = new Point (100,200);
      p2.Move (50,50);
      Console.WriteLine (p2);
      Point.Info ();
   }
}
```

Results:

```
The number of points is 0
(5,5)
(150,250)
The number of points is 2
```

Extension Methods

Extension method

- **add methods to existing types without creating a new derived type, or recompiling**
- **special kind of static method**
- **called by using instance method syntax**

Defining and calling the extension method

1. **Define a static class to contain the extension method. The class must be visible to client code.**
2. **Implement the extension method as a static method with at least the same visibility as the containing class.**
3. **The first parameter of the method specifies the type that the method operates on; it must be preceded with the this modifier.**
4. **In the calling code, add a using directive to specify the namespace that contains the extension method class.**
5. **Call the methods as if they were instance methods on the type. The first parameter is not specified by calling code because it represents the type on which the operator is being applied. You only have to provide arguments for parameters 2 through *n*.**

```
using System;
namespace StringExtensionMethods
{   public static class StringExtension
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace ExtensionMethods
{   using StringExtensionMethods;
    class Program
    {   static void Main(string[] args)
        {
            string s = "This a test of  extension   methods";
            int count = s.WordCount();
            Console.WriteLine("Word count = " + count);
        }
    }
}
```

Word count = 6

Binding Extension Methods at Compile Time

- **Extension methods extend a class or interface, but not to override them.**
- **An extension method with the same name and signature as an interface or class method will never be called.**
- **At compile time, the compiler it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds.**

```
using System;
namespace ExtensionMethods
{   using StringExtensionMethods;
    class Program
    {
        private String str;
        public Program(String str)
        {
            this.str = str;
        }
        public int WordCount()
        {   Console.WriteLine("Instance method call");
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;;
        }
        static void Main(string[] args)
        {   Program s = new Program("This a test of  extension   methods");
            int count = s.WordCount();
            Console.WriteLine("Word count = " + count);
        }
    }
}
```

Instance method call
Word count = 6

Abstract Classes

Abstract class – defines features of derived, non-abstract classes

- **cannot be instantiated**
- **abstract members are defined using the abstract keyword**
- **abstract methods have no implementation and the derived classes must implement all abstract methods using override modifier**
- **provide a common definition of a base class that multiple derived classes can share**

Example:

```
using System;
abstract class RoundShape  // Abstract class
{
   protected class Center     // Nested class
   {   public int x;
       public int y;
   }
   protected Center c = new Center();
   protected float radius;
   abstract public float Area();      // Abstract method
   public RoundShape (int x, int y, float r)
   {    c.x = x;
        c.y = y;
        radius = r;
   }
}
```

```
class Circle : RoundShape
{
   public Circle(int x, int y, float r) : base(x, y, r) { }
   public override float Area()           // Implementation of Area
   {
       return (float)(Math.PI*Math.Pow((double)radius, 2.0));
   }
}
class Sphere : RoundShape
{
   public Sphere(int x, int y, float r) : base(x, y, r) { }
   public override float Area()           // Implementation of Area
   {
       return (float)(4*Math.PI*Math.Pow((double)radius, 2.0));
   }
}
```

```
class Shape
{
   static void Main()
   {
       RoundShape shape;
       shape = new Circle (5, 5, 10.0F);
       Console.WriteLine ("The area of the circle is " +
                          shape.Area());
       shape = new Sphere (5, 5, 10.0F);
       Console.WriteLine ("The area of the sphere is " +
                          shape.Area());
   }
}
```

Results:

The area of the circle is 314,1593
The area of the sphere is 1256,637

Properties

Properties – referred to as smart fields on the client side and have the same capabilities as accessors.

1. **Defining a Property – consists of a field declaration and accessors used to modify that field's value**

```
[attributes] [modifiers] <type> <property_name>
{
   [set
    {  <accessor_body>  }
   ]
   [get
    {  <accessor_body>  }
   ]
}
```

- **must have one of set or get method**
- **read-write property – both set and get methods are defined**
- **read-only property – only get method is defined**
- **write-only property – only set method is defined**
- **can't be used as parameters to methods (they are not fields)**
- **can be defined with the static modifier, but can't be combined with the virtual, abstract, override, because they are used only for instance members**
- **afford advantage over using accessors**

Example:

```
using System;
class Address
{  protected string city;
   public string City                          // Read-only property
   {  get  {  return city;  }  }
   protected int telCode;
   public int TelCode                  // Read-write property
   {  get  {  return telCode;  }
      set
      {  // Validate value against some data store
         telCode = value;
         // Update city based on validated telCode
         if (telCode == 2)
            city = "Sofia";
      }
   }
}
```

```
class PropertyApp
{
   static void Main()
   {
      Address addr = new Address();
      addr.TelCode = 2;
      Console.WriteLine
            ("The city with telephone code {0} is {1}.",
              addr.TelCode, addr.City);
   }
}
```

Results:

The city with telephone code 2 is Sofia.

The compiler emits items for each property:

- get **accessor method:** get_City, get_TelCode
- set **accessor method:** set_TelCode
- **Property definitions (always):**

```
.property instance string City
{
   .get instance string Address::get_City()
} // end of property Address::City
.property instance int TelCode
{
   .get instance int Address::get_TelCode()
   .set instance void Address::set_TelCode (int)
} // end of property Address::TelCode
```

2. Inheriting Properties

- **Overriding Inherited Properties – the virtual modifier for the property that can be overridden, the override modifier on the derived class's implementation the inherited property**

Example:

```
using System;
class Address
{
   protected string city;
   public string City
   {
      get
      {
        return city;
      }
   }
```

```
   protected int telCode;
   public virtual int TelCode
   {
      get   {   return telCode;   }
      set
      {   telCode = value;
         // Update city based on validated telCode
         if (telCode == 2)
            city = "Sofia";
      }
   }
}
class FullAddress : Address
{   private string state;
   public string State
   {
      get   {   return state;   }
   }
```

```
   public override int TelCode      // Override the inherited
   {                                // property
      set
      {
         telCode = value;
         // Update city based on validated telCode
         if (telCode == 3592)
         {
            city   = "Sofia";
            state = "Bulgaria";
         }
      }
   }
}
```

```
class PropertyApp
{
   static void Main()
   {
      FullAddress addr = new FullAddress();
      addr.TelCode = 3592;
      Console.WriteLine("Code: {0}, City: {1}, State: {2}.",
                        addr.TelCode, addr.City, addr.State);
   }
}
```

Results:

Code: 3592, City: Sofia, State: Bulgaria.

- **Enforcing Property Implementation via** abstract **Properties**

```
abstract class <abstract_base_class_name>
{
    public abstract <type> <abstract_property_name>
    {
        get;
        set;
    }
}
```

Example:

```
using System;
abstract class Employee
{   protected int id;
    public int Id
    {
      get   {   return id;   }
    }
    protected int hoursWorked;
    protected double hourlyCost;
    public abstract double HourlyCost
    {
       get;
    }
    protected Employee (int id, int hoursWorked)
    {   this.id = id;
       this.hoursWorked = hoursWorked;
    }
```

```
    public override string ToString()
    {
       return "Employee (id = " + id + ") costs " + HourlyCost +
              " per hour.";
    }
}
class ContractEmployee : Employee
{   protected double hourlyWage;
    public override double HourlyCost
    {
      get   {   return hourlyWage;   }
    }
    public ContractEmployee (int id, int hoursWorked,
                             double hourlyWage) : base(id, hoursWorked)
    {
       this.hourlyWage = hourlyWage;
    }
}
```

```
class SalariedEmployee : Employee
{
    protected double salary;
    public override double HourlyCost
    {
       get
       {
         return salary / hoursWorked;
       }
    }
    public SalariedEmployee(int id,int hoursWorked,double salary)
                    : base(id,hoursWorked)
   {
       this.salary = salary;
    }
}
```

```
class OverrideProperties
{
    static void Main()
    {
       Employee e;
       e = new ContractEmployee (1, 40, 20);
       Console.WriteLine(e);

       e = new SalariedEmployee (2, 160, 400);
       Console.WriteLine(e);
    }
}
```

Results:

```
Employee (id = 1) costs 20 per hour.
Employee (id = 2) costs 2,5 per hour.
```

3. Advanced Use of Properties

- **provide high level of abstraction – the client doesn't need to know if an accessor exists for the member being accessed**
- **provide a generic means of accessing class members by using the standard syntax**

  object.field
- **guarantee the additional processing of a particular field that is modified or accessed**

4. Auto-Implemented Properties

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects.

The compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors.

```
using System
public class PhoneEntry
{
  public String Name { get; set; }
  public long Phone { get; set; }
  public static void Main()
  {
    PhoneEntry p = new PhoneEntry ();
    p.Name = "Maria";
    p.Phone = 359888881000;
    Console.WriteLine(p.Name + " " + p.Phone);
  }
}
```

The compiler automatic emits a field of type String (according to the type of the property Name), a field of type long (according to the type of the property Phone) and methods get_Name, set_Name, get_Phone and set_Phone, that get/set the field values.

```
using System;
class Rational
{    public int Nominator { get; set; }
     public int Denominator { get; set; }
     public Rational(int nominator, int denominator)
     {
        Nominator = nominator;
        Denominator = denominator;
     }
     public Rational() { }
     public override string ToString()
     { return Nominator + "/" + Denominator; }
  }
  class Program
  {
     static void Main(string[] args)
     {
        Rational r = new Rational(1, 5);
        float number = (float)r.Nominator / (float)r.Denominator;
        Console.WriteLine(r + " = " + number);
     }
  }
```

1/5 = 0.2

5. Object Initializers

```
Rational r = new Rational() { Nominator=1, Denominator=5 };
```

that is identical to:

```
Rational r = new Rational() ;
r.Nominator=1;
r.Denominator=5;
```

We can omit the parentheses:

```
Rational r = new Rational { Nominator=1, Denominator=5 };
```

In one statement we can construct an Rational object, called its constructor, initialized two public properties and call ToString and ToUpper on the resulting expression – code in expression context.

```
String s=new Rational{Nominator=1, Denominator=5}.ToString().ToUpper();
```

6. Anonymous Types

var o = new { $property_1$ = $expression_1$, ..., $property_N$ = $expression_N$ }

The compiler infers the type of each $expression_i$, creates: private fields of these types, public read-only properties for each of the fields, a constructor that accepts all these expresions, and overrides: Equals, GetHashCode and ToString methods.

```
var r1 = new { Nominator=1, Denominator=5 };
Console.WriteLine(r1);              // {Nominator=1, Denominator=5 }
```

The compiler can generate anonymous type where it can infer the property names and types from variables:

```
int Nominator=1;
int Denominator=5;
var r2 = new {Nominator, Denominator };
Console.WriteLine(r2);              // {Nominator=1, Denominator=5 }
Console.WriteLine(r1.Equals(r2));                    //  True
```

Operator Overloading

Operator overloading

- **allows to be redefined existing operators so that one or both of the operands are of a class or struct type**
- **another means of calling a method**
- **aids abstraction – one of the most important aspects of object-oriented programming**

Operator Overloading Syntax

op – **overloading operator**

```
public static <return_type_value> operator op
                                (<argument1>[,<argument2>])
{
    <operator-overloading-body>
    return return_value;
}
```

- **must be defined as** public **and** static
- <return_type_value> – **any type (commonly** class **or** struct**, Boolean value for** true **and** false **operator)**
- **for unary operator –** <argument1> **of type** class **or** struct**;**
- **for binary operator –** <argument1> **of type** class **or** struct**,** <argument2> **– of any type**

Rules and Restrictions

1. **There are two categories of operator overloading:**
   - **unary** + - ~ ++ -- true false
   - **binary** + - * / % & | ^ << >> == =! > < >= <=
2. **Can't be overloaded the operators:**
   - , [] () && || ?:
   - **undefined operators in C# (for example **)**
   - **defined at run time – member access (. dot), member invocation, assignment (=) and** new
3. **Overloaded by pair (if one operator is overloaded, the other must be overloaded as well):**
   - == **and** != **(must also be overloaded the** Equals **and** GetHashCode **methods)**
   - < **and** >

4. **The assignment operator can't be overloaded, when a binary operator is overloaded, its compound assignment operator is implicitly overloaded – for example, if the overloaded operator is** +**, the** += **operator is implicitly overloaded.**

Example: Operator + **overloading in the** Rational **class**

```
using System;
class Rational
{
   private int numerator;
   private int denominator;
   public Rational (int numer, int denom)
   {
      numerator = numer;
      denominator = denom;
      Reduce();
   }
```

```
   private void Reduce()
   {
      int common = Gcd (Math.Abs(numerator), denominator);
      numerator /= common;
      denominator /= common;
   }
   private int Gcd (int n1, int n2)    // Greater common divisor
   {
      while (n1 != n2)
        if (n1 > n2)
          n1 -= n2;
        else
          n2 -= n1;
      return n1;
   }
```

```
   // Overloading + operator
   public static Rational operator+ (Rational op1, Rational op2)
   {
      int commonDenominator =
                                  op1.denominator*op2.denominator;
      int sum = op1.numerator*op2.denominator +
                                  op1.denominator*op2.numerator;
      return new Rational (sum, commonDenominator);
   }
   public override string ToString()
   {
      return numerator + "/" + denominator;
   }
```

```
    // Override explicit convertion from Rational to float
    public static explicit operator float(Rational op)
    {
        return (float)op.numerator / op.denominator;
    }

    // Override implicit convertion from Rational to double
    public static implicit operator double(Rational op)
    {
        return (double)op.numerator / op.denominator;
    }
}
```

```
class TestRational
{   static void Main()
    {   Rational x, y, z;
        x = new Rational (1, 4); y = new Rational (1, 3);
        z = x + y;
        Console.WriteLine (x + "+" + y + "=" + z);
        Console.Write (z);
        z += y;
        Console.WriteLine ("+=" + y + "=" + z);
        float f = (float)z; Console.WriteLine(f);
        double r = z; Console.WriteLine(r);
    }
}
```

Results:

```
1/4+1/3=7/12
7/12+=1/3=11/12
0.5833333
0.583333333333333
```