Classes & Objects

C# Programming

Object-Oriented Programming Concepts

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

- Grady Booch

A bulb:

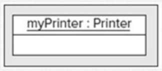
1. It's a real-world thing. → object
2. Can be switched on to generate light and switched off. → methods
3. It has real features like the glass covering, filament and holder. → member variables
4. It also has conceptual features like power.
5. A bulb manufacturing factory produces many bulbs based on a basic description / pattern of what a bulb is. → class

What Is an Object?

- An object is a building block of an OOP application.
- It encapsulates part of the application, which may be a process, a chunk of data, or a more abstract entity.
- Objects in C# are created from the types, just like the variables.
- The type of an object is known by a special name in OOP, its class.
- Think of a class as the template for the car, or perhaps the plans used to build the car. The car itself is an instance of those plans, so it could be referred to as an object.

How it is in UML?

- UML is designed for modelling applications from the objects that build them to the operations they perform to the use cases that are expected.
- Inside UML the class name is shown in the top section of the box.
- For example in a UML representation of an instance of Printer class called myPrinter, the instance name is shown first in the top section, followed by the name of its class. The two names are separated by a colon.



Class

- A class is a user defined type which comprises Data and Functionality both unlike to the struct keyword of 'C'.
- It is a template / skeleton / blueprint for creating an object.
- Class in C# is created using the class keyword.
- Objects in C# are created from classes, just like the variables.
- You can use class definitions to instantiate objects, which means creating a real, named instance of a class.

Class Definition in C#

```
class MyClass
{
        // Class members.
}
```

- This code defines a class called MyClass. Once you have defined a class, you are free to instantiate it anywhere else in your project that has access to the definition.
- By default, classes are declared as internal, meaning that only code in the current project will have access to them.

Access Modifiers

MODIFIER	DESCRIPTION
none or internal	Class is accessible only from within the current project.
public	Class is accessible from anywhere
abstract or internal abstract	Class is accessible only from within the current project, and cannot be instantiated, only derived from
public abstract	Class is accessible from anywhere, and cannot be instantiated, only derived from
sealed or internal sealed	Class is accessible only from within the current project, and cannot be derived from, only instantiated
public sealed	Class is accessible from anywhere, and cannot be derived from, only instantiated

Object Creation and Instantiation

- When an object is created all the variables (value/ reference types) are allocate the memory in heap as a single unit and default values are set to them based on their types.

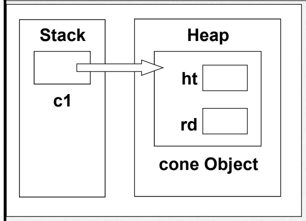
```
using System;
public class App
{
   class MyClass
   {
      // Class members.
   }
   static void Main()
   {
      MyClass obj1;
      obj1 = new MyClass();
   }
}
```

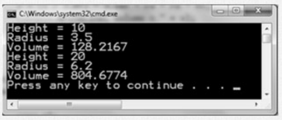
- Here obj1 is a reference of a type MyClass
- It is not an object and till now no such memory allocation for an object is done
- To allocate a memory we have to use a 'new' keyword

Class an Example

```
using System;
namespace Shape
{
    class cone
    {
        float ht, rd;
        public void set(float h, float r)
        {
            ht = h; rd = r;
        }
        public void display()
        {
            Console.WriteLine("Height = " + ht);
            Console.WriteLine("Radius = " + rd);
        }
        public void volume()
        {
            float v;
            v = (1 / 3.0f) * 3.14f * rd * rd * ht;
            Console.WriteLine("Volume = " + v);
        }
    }
```

Class an Example Cont…





```
Height = 10
Radius = 3.5
Volume = 128.2167
Height = 20
Radius = 6.2
Volume = 804.6774
Press any key to continue . . . _
```

```
class Class1
{
  static void Main(string[] args)
  {
      cone c1;
      c1 = new cone();
      cone c2;
      c2 = new cone();
      c1.set(10.0f, 3.5f);
      c1.display();
      c1.volume();
      c2.set(20.0f, 6.2f);
      c2.display();
      c2.volume();
   }
  }
}
```

Class Members

- The variables and functions defined inside a class are called members of the class.
- Types of members
  - Data members/ Fields
  - Member functions
    - Methods
    - Constructors
    - Finalizers
    - Operators
    - Properties
    - Indexer

Scope of variable

- Scope of variable can be largely divided into three categories
  - Local
    - Declarations inside method; accessible only inside the method
    - Local scoped variables must be initialized before use
  - Class
    - Declarations are made outside a method and inside a class; accessible only inside the class
    - Class scoped variables are automatically initialized to their default values.
    - There are 2 types of variables are defined in class
      - Instance variables
      - Static/class variables
  - Namespace
    - Within a namespace, you can declare another namespace, class, interface, struct, enum, delegate only

Member variable default values

- Member variables are automatically assigned a default value.
  - bool → false
  - Integer types → 0
  - Floating point types → 0.0
  - char → '\0'
  - string and references → null
- Note that the compiler requires the local variables (variables declared within a method) EXPLICITLY initialized before use!!

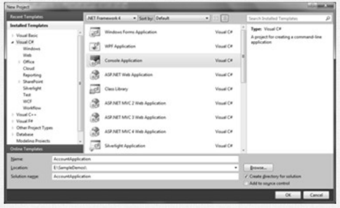
Member and class visibility

- Member Visibility
  - public
    - Accessible from anywhere
  - private
    - Accessible only from within a class
    - Unmarked members are private by default
  - internal
    - Accessible only within files in the same assembly
  - protected
  - protected internal

  Applicable when inheriting. More in inheritance session
- Top-level Class Visibility – public or internal

Account Class

- Let's walk through a live example of an OOPS, that will clearly explain object creation and also the use of different member of the class.
- Step 1: Create a new project (File → New Project) Console Application.


New Project
Recent Templates
Installed Templates
Visual Basic
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Visual C++
Visual F#
Other Project Types
Database
Modeling Projects
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Class Library Visual C#
ASP.NET MVC 2 Web Application Visual C#
ASP.NET MVC 3 Web Application Visual C#
ASP.NET MVC 4 Web Application Visual C#
Silverlight Application Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating a command-line application

Name: AccountApplication

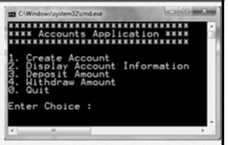
Location: E:\SampleDemos\

Solution name: AccountApplication

☑ Create directory for solution

□ Add to source control

Account Application CUI



```
*******************************
**** Accounts Application ****
*******************************

1. Create Account
2. Display Account Information
3. Deposit Amount
4. Withdraw Amount
0. Quit

Enter Choice :
```

```
using System;
namespace AccountsDemoApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int ch;
            Account a = null;
            do
            {
                Console.WriteLine("*******************************");
                Console.WriteLine("**** Accounts Application ****");
                Console.WriteLine("*******************************");
                Console.WriteLine("1. Create Account");
                Console.WriteLine("2. Display Account Information");
                Console.WriteLine("3. Deposit Amount");
                Console.WriteLine("4. Withdraw Amount");
                Console.WriteLine("0. Quit");
                Console.WriteLine("Enter Choice :");
                ch = int.Parse(Console.ReadLine());
```

Account Application CUI

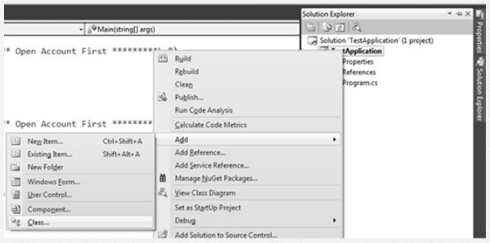
```
switch (ch)
{
   case 1:
    Console.WriteLine("Enter Account Id:");
    int Id = int.Parse(Console.ReadLine());
    Console.WriteLine("Enter Name:");
    string Name = Console.ReadLine();
    Console.WriteLine("Enter Opening Balance:");
    decimal Balance = decimal.Parse(Console.ReadLine());
    a = new Account();
    a.CreateAccount(Id, Name, Balance);
    break;
   case 2:
    if (a != null)
        a.DisplayInfo();
    else
        Console.WriteLine("********* Open Account First *********\n");
    break;
   case 3:
    if (a != null)
        a.Deposit(5000);
    else
        Console.WriteLine("********* Open Account First *********\n");
    break;
```

Account Application CUI

```
     case 4:
      if (a != null)
        a.Withdraw(5000);
      else
        Console.WriteLine("********* Open Account First *********\n");
      break;
     case 0:
       break;
     default:
       Console.WriteLine("********* Wrong Choice *********\n");
       break;
     }
   }while (ch != 0);
   Console.WriteLine("********* Thanks for Using Banking Operations
                     *********\n");
  }
 }
}
```

Account Application Account Class

- In Solution Explorer right click on project and add new class using



- One more .cs file with Account will be added.

Create Account Class

```
using System;
namespace AccountsDemoApp
{
    class Account
    {
        public int Id;
        public string Name;
        public decimal Balance;
        public Account()
        {
            Console.WriteLine("Object Created");
        }
        ~Account()
        {
            Console.WriteLine("Object Destroyed");
        }
        public void CreateAccount(int id, string name, decimal balance)
        {
            Id = id;
            Name = name;
            Balance = balance;
        }
```

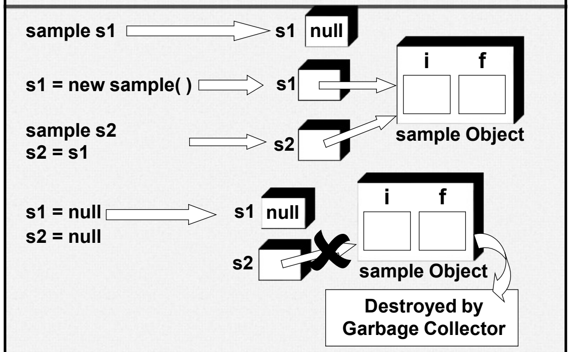
Adding Methods in Accounts Class

```
        public void DisplayInfo()
        {
            Console.WriteLine("***** Account Information *****");
            Console.WriteLine("Account Id:" + Id);
            Console.WriteLine("Name:" + Name);
            Console.WriteLine("Opening Balance:" + Balance );
        }
        public void Deposit(decimal amount)
        {
            Balance += amount;
        }
        public void Withdraw(decimal amount)
        {
            if (Balance - amount < 500)
                throw new ApplicationException("Insufficient Balance");
            else
                this.Balance -= amount;
        }
    }
}
```

Add more to an Application

```
…
…
Console.WriteLine("5. Destroy An Object");
Console.WriteLine("6. Invoke Garbage Collector");
Console.WriteLine("7. Temp Object");
Console.WriteLine("8. Get All Generations");
…
…
case 5:
    a = null;
    break;
case 6:
   GC.Collect();
   break;
case 7:
   Account a1 = new Account();
   a = a1;
   break;
case 8:
   Console.WriteLine(GC.GetGeneration(a).ToString());
   break;
```

References



sample s1 → s1 null

s1 = new sample( ) → s1 → i f

sample s2
s2 = s1 → s2 → sample Object

s1 = null → s1 null
s2 = null → s2 → i f sample Object

Destroyed by Garbage Collector

What we added?

- **Destroy An Object**: If a reference variable is not initialized i.e referring to a null, then trying to access any member using it, will throw a runtime exception i.e. "NullReferenceException".
- In fact we cannot destroy an object. Objects are destroyed by Garbage Collector only. If Garbage Collector finds any object unreferenced in memory then it destroys an object.
- **Invoke Garbage Collector**: Used to force the activation of Garbage Collector for garbage collection process.
- The GC class's Collect() method is use to force system to collect garbage i.e. null referenced object will be collected by Garbage Collector

```
GC.Collect();
```

What we added?

- **Temp Object**: it will create another local reference variable of type Account named a1 referring to new Object of Account class.

```
Account a1 = new Account();
a = a1;
```

- a = a1; Copies the value of a (reference to the object) into a1 and thus both "a1" and "a" refers to the same object.
- **Get All Generations:** Determine the age of an object and returns the current generation number of the specified object. Determine which generation "a" object is stored in.

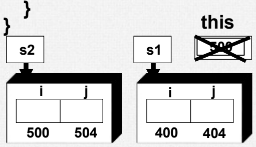
```
GC.GetGeneration(a).ToString()
```

this Pointer

```
using System;
class sample
{
    int i ;
    float j ;
    public void setdata ( int ii,
                          float  jj )
    {
        this.i = ii ;
        this.j = jj ;
    }
}
```

□ this is Optional
□ this cannot be modified

```
class Class1
{
  static void Main( strings [ ]
                          args  )
  {
        sample s1 = new
                     sample( ) ;
        s1.setdata ( 10, 20.4f )  ;
        sample s2 = new
                     sample( ) ;
        s2.setdata ( 5, 10.6f ) ;
  }
}
```



Why this?

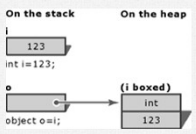
```
using System;
class sample
{
   int i ;
   float j ;
   public void setdata (  int i ,
                          float j )
   {
     this.i = i ;
     this.j = j ;
   }
   public void display ( )
   {
        C.W ( i + "  " + j ) ;
   }
}
```

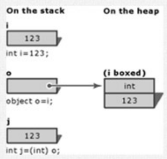
```
class Class1
{
        static void Main ( string [ ]
                               args )
        {
           sample s1 = new
                       sample( ) ;
           s1.setdata ( 10, 5.4f ) ;
           s1.display ( ) ;
        }
   }
```

10 5.4

Boxing / Unboxing

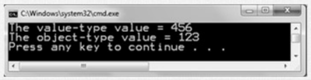
- Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type.
- When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap.
- Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit.





Boxing Unboxing Example

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

       // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i does not effect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
```



```
The value-type value = 456
The object-type value = 123
Press any key to continue . . .
```

About Example

- This example converts an integer variable i to an object o by using boxing.
- Then, the value stored in the variable i is changed from 123 to 456.
- The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

Properties

- Properties are named members of classes (structs, and interfaces) that provide a flexible mechanism to read, write, or compute the values of private fields through accessors.

  Syntax:

  ```
  Access-specifier type name {
  get{}
  set{}
  }
  ```
- The properties are accessed using their names.
- The implicit parameter value is used in setting or getting.

Syntax

```
private <Datatype> _<propertyname>;
public <Datatype> <PropertyName>
{
   [private] get
   {
         return _<propertyname>;
   }
   [private] set
   {
         _<propertyname> = value;
   }
}
```

Encapsulate Account Class

- Add properties for Name, Balance and Id with following conditions:
  - Balance should be ReadOnly.
  - Name should accept greater than or equal to 15 characters only
- Edit the code in the Account Class

Partial Access and Validation

```
private decimal _Balance;
public decimal Balance
{
   get
   {
      return _Balance;
   }
   set
   {
      _Balance = value;
   }
}
```

```
private string _Name;
public string Name
{
	get
	{
		return _Name;
	}
	set
	{
		if (value.Length > 8)
			throw new
ApplicationException("Name cannot be > 8
characters");
		_Name = value;
	}
}
```

The Life Cycle of an Object

- Every object has a clearly defined life cycle.
- This life cycle includes two important stages:
  - **Construction:** When an object is first instantiated it needs to be initialized. This initialization is known as construction and is carried out by a constructor function, often referred to simply as a constructor for convenience.
  - **Destruction:** When an object is destroyed, there are often some clean-up tasks to perform, such as freeing memory. This is the job of a destructor function, also known as a destructor.

Constructors

- Basic initialization of an object is automatic.
- If you want to perform initialization of data stored in an object during object creation you can use constructor.
- All class definitions contain at least one constructor.
- A class definition might also include several other constructor methods with parameters, known as parameterized constructors.
- Constructors are called using the new keyword.

Working with Constructors

```
using System;
namespace ConstructorDemo
{
    class cone
    {
        float ht, rd;
        public cone(float h, float r)
        {
            ht = h; rd = r;
        }
        public void display()
        {
            Console.WriteLine("Height = " + ht);
            Console.WriteLine("Radius = " + rd);
        }
        public void volume()
        {
            float v;
            v = (1 / 3.0f) * 3.14f * rd * rd * ht;
            Console.WriteLine("Volume = " + v);
        }
    }
```

Working with Constructors

```
class Program
    {
        static void Main(string[] args)
        {
            cone c1;
            c1 = new cone(10.0f, 3.5f);
            cone c2;
            c2 = new cone(20.0f, 6.2f);
            c1.display();
            c1.volume();
            c2.display();
            c2.volume();
        }
    }
}
```

A constructor is a member method of a class which is automatically executed / called as soon as the object of that class is created and thus it is ideally used for initializing the field members of the object.

Point to be Considered

- A constructor has same name as the class in C# and doesn't have a return type not even void.
- A constructor without a parameter is called default constructor and the one with parameters is called as Parameterized constructor.
- If a class doesn't have any form of constructor, a public default constructor is automatically added to it by the language compiler.
- Copy Constructor is used to create a new object by duplicating the state of an existing object and this is done by copying the data members of existing object in new object data members.

Syntax : Defining Constructors

```
<Class name>()
{
  //default constructor
}
<Class name>(<datatype> p1, …)
{
//parameterized constructor
}
<Class name>(<Class Name> <parname>) : this(. . .)
{
//copy constructor
}
```

Constructor Chaining

```
class Account
{
    public int Id;
    public string Name;
    public decimal Balance;
    public Account()
    {       }
    public Account(int Id, String name, decimal balance): this()
    {
        this.Id = Id;
        this.Name = name;
        this.Balance = balance;
    }
    public Account(Account a) : this(a.Id, a.Name, a.Balance)
    {
        // this.Id = Id;
        //this.Name = a.Name;
        //this._Balance = a.Balance;
    }
}
```

Working with Destructor

- A destructor is used to release the resources that an object is holding.
- When an object is ready to be destroyed Finalize method / Destructor is called on that object.
- Syntax

  ```
  ~<classname>()
  {
  }
  ```

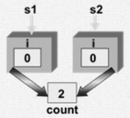
Initializing Data Member

- Elegancy of C# is that we can even initialize data members of a class where they are declared.

```
using System;
namespace Shape
{
    class cone
    {
        float ht = 5.3f;
        float rd = 2.4f;

        // Methods
    }
}
```

Static Data

```
using System;
class sample
{
    int i;
    static int count = 0;
    public sample()
    {
        i = 0;
        count++;
    }
    static void showcount()
    {
        Console.WriteLine(count);
    }
}
```



```
class Program
{
    static void Main(string[]
args)
    {
        sample s1 = new sample();
        sample.showcount();
        sample s2 = new sample();
        sample.showcount();

    }
}
```

Static functions can access only static data

Static Property-Example

```
using System;
class Circle
{
    private static double PI;
    public static double pi
    {
        get { return PI; }
        set { PI = value; }
    }
    public Circle(int n)
    {
        pi = n;
    }
}
class Program
{
    static void Main()
    {
        Circle c = new Circle(4);
        Circle.pi = 3;
        Console.WriteLine(Circle.pi);
    }
}
```

Static Constructor

```
using System;
namespace StaticDemo
{
    class sample
    {
        public static int y;
        public static int m;
        public static int d;

        static sample()
        {
            DateTime dt;
            dt = DateTime.Now;
            y = dt.Year;
            m = dt.Month;
            d = dt.Day;
        }
```

```
     public static void showdate()
     {
       Console.WriteLine("Year:" + y);
       Console.WriteLine("Month:"+ m);
       Console.WriteLine("Day:" + d);
     }
    }
    class Class1
    {
       static void Main(string[] args)
       {
          sample.showdate();
       }
     }
    }
```

Static Constructor

- The static constructor is invoked when the class is first loaded.
- If the object of the class is created the static constructor is executed before the simple constructor is called.
- The static constructor is always public and access modifiers are not allowed on it.
- We cannot access non-static or instance member variables or methods in this constructor.
- The static constructors are always zero-argument constructors.

Static Classes

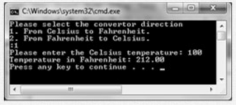
- A class can be declared static.
- There are two key features of a static class.
  - First, no object of a static class can be created.
  - Second, a static class must contain only static members.
- Within the static class, all members must be explicitly specified as static.
- Making a class static does not automatically make its members static.
- A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields.

Static Class & Method

```
using System;
namespace CSProgExample
{
    public static class TemperatureConverter
    {
        public static double CelsiusToFahrenheit
                         (string temperatureCelsius) {
            double celsius = Double.Parse(temperatureCelsius);
            double fahrenheit = (celsius * 9 / 5) + 32;
            return fahrenheit;
        }
        public static double FahrenheitToCelsius
                         (string temperatureFahrenheit) {
            double fahrenheit = Double.Parse(temperatureFahrenheit);
            double celsius = (fahrenheit - 32) * 5 / 9;
            return celsius;
        }
    }
    class TestTemperatureConverter
    {
        static void Main() {
            Console.WriteLine("Please select the convertor direction");
            Console.WriteLine("1. From Celsius to Fahrenheit.");
            Console.WriteLine("2. From Fahrenheit to Celsius.");
            Console.Write(":");
            string selection = Console.ReadLine();
```

Example cont.

```
            double F, C = 0;
            switch (selection)
            {
                case "1":
                    Console.Write("Please enter the Celsius temperature: ");
                    F = TemperatureConverter.CelsiusToFahrenheit
                                                    (Console.ReadLine());
                    Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                    break;
                case "2":
                    Console.Write("Please enter the Fahrenheit temperature: ");
                    C = TemperatureConverter.FahrenheitToCelsius
                                                    (Console.ReadLine());
                    Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                    break;
                default:
                    Console.WriteLine("Please select a convertor.");
                    break;
            }
        }
    }
}
```



```
Please select the convertor direction
1. From Celsius to Fahrenheit.
2. From Fahrenheit to Celsius.
:1
Please enter the Celsius temperature: 100
Temperature in Fahrenheit: 212.00
Press any key to continue . . . _
```

Object Initializers

- Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to explicitly invoke a constructor.
- The compiler processes object initializers by first accessing the default instance constructor, and then by processing the member initializations.
- Therefore, if the default constructor is declared as private in the class, object initializers that require public access will fail.

Using Object Initializers

```
//The following example shows how to initialize
//a new StudentName type by using object initializers.
using System;
public class StudentName
{
    // The default constructor has no parameters. The default
    // constructor is invoked in the processing of object
    // initializers. You can test this by changing the access
    // modifier from public to private.
    // The declarations in Main that use object
    // initializers will fail.
    public StudentName() { }

    // The following constructor has parameters
    // for two of the three  properties.
    public StudentName(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }
    // Properties.
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
```

Example Cont.

```
    public override string ToString()
    {
        return FirstName + "  " + ID;
    }
}
public class Program
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor
        //that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");
        // Make the same declaration by using a collection
        // initializer and sending arguments for the first
        // and last names. The default constructor is invoked
        // in processing this declaration, not the constructor
        // that has two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
        };
```

Example Cont.

```
        // Declare a StudentName by using a collection
        // initializer and sending  an argument for only
        // the ID property. No corresponding constructor is
        // necessary. Only the default constructor is used
        // to process object  initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };
        // Declare a StudentName by using a collection initializer and
        // sending arguments for all three properties.
        // No corresponding constructor is defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };
        System.Console.WriteLine(student1.ToString());
        System.Console.WriteLine(student2.ToString());
        System.Console.WriteLine(student3.ToString());
        System.Console.WriteLine(student4.ToString());
    }
}
```

Auto Implemented Properties

Access-specifier type name { get; set; }

- Object initializers are automatically implemented properties where both get and set accessors are defined automatically.
- We cannot have one accessor automatically implemented and the other explicitly implemented.
- Also, in this case, 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 Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public static void Main()
    {
        Point p = new Point { X = 10, Y = 20 };
        Console.WriteLine("({0}, {1})", p.X, p.Y);
    }
}
```

Read Only

- C# provides a keyword readonly which is used for variables which are to be initialized at the time of building the objects.

```
using System ;
class Hotel
{
    public const int rate = 4000;
    public readonly string name;
    public Hotel(string n)
    {
        name = n;
    }
```

```
class Class1
{
   static void Main(string[] args)
   {
      Hotel h;
      h = new Hotel("Tuli");
      Console.WriteLine(h.name);
      h.name = "CP"; //error
      Console.WriteLine(Hotel.rate);
      Hotel.rate = 1500;  //error
   }
```

Tips

□ RO can be initialized in ctor

□ RO values can be diff for diff objects

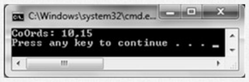
□ constants are implicitly static

Partial Classes and Methods

- It is possible to split the definition of a class or a struct, an interface or a method over two or more source files.
- Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.
- The partial keyword indicates that other parts of the class, struct, or interface can be defined in the namespace.
- All the parts must use the partial keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as public, private etc.

Partial Class Example

```
using System;
namespace CSProgExample
{
    public partial
              class CoOrds
    {
        private int x;
        private int y;
        public CoOrds
              (int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }
```



```
CoOrds: 10,15
Press any key to continue . . . _
```

```
public partial class CoOrds
    {
        public void PrintCoOrds()
        {
            Console.WriteLine
           ("CoOrds: {0},{1}", x, y);
        }
    }
    class TestCoOrds
    {
        static void Main()
        {
            CoOrds myCoOrds
               = new CoOrds(10, 15);
            myCoOrds.PrintCoOrds();
        }
    }
}
```

Partial Methods

- A partial class or struct may contain a partial method.
- One part of the class contains the signature of the method.
- An optional implementation may be defined in the same part or another part.
- If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Rules for Partial Methods

- Partial methods are indicated by the partial modifier.
- Partial methods must be private.
- Partial methods must return void.
- Partial methods must only be declared within partial classes.
- Partial methods do not always have an implementation.
- Partial methods can be static and generic.
- Partial methods can have arguments including ref but not out.
- You cannot make a delegate to a partial method.

Partial Method in Use

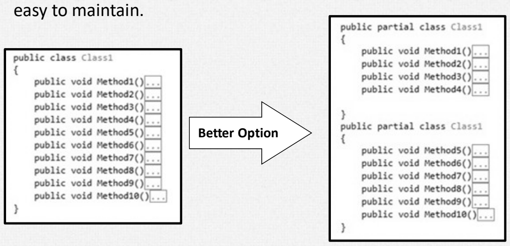
- As shown below a method of class can be separated into two different code files in the same project.
- During compile time, these files get compiled into a single class.

```
using System;
namespace CSharpProg
{
    public partial class Customer
    {
        public void Add()
        {
        }
        //Definition
        partial void Validate();
    }
}
```

```
using System;
namespace CSharpProg
{
    public partial class Customer
    {
        public void Update()
        {
            Validate();
        }
        //Implementation
        partial void Validate()
        {
        }
    }
}
```

Another Feature

- If you have large classes with lots of methods as shown in the figure, it’s a bit of a pain to maintain those classes.
- By using partial classes, you can split them into physical files as shown in the below figure, thus making your project better and easy to maintain.

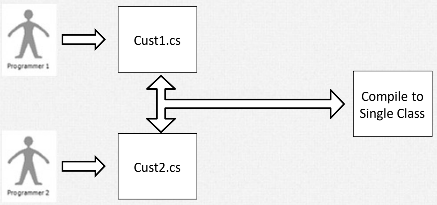


```
public class Class1
{
    public void Method1()...
    public void Method2()...
    public void Method3()...
    public void Method4()...
    public void Method5()...
    public void Method6()...
    public void Method7()...
    public void Method8()...
    public void Method9()...
    public void Method10()...
}
```

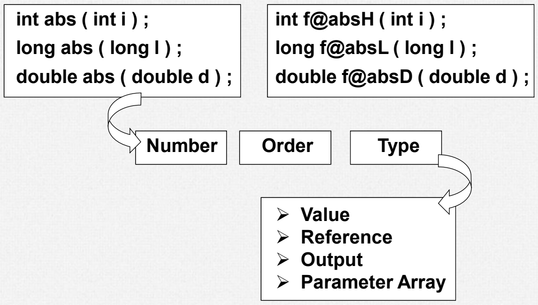
```
public partial class Class1
{
    public void Method1()...
    public void Method2()...
    public void Method3()...
    public void Method4()...

}
public partial class Class1
{
    public void Method5()...
    public void Method6()...
    public void Method7()...
    public void Method8()...
    public void Method9()...
    public void Method10()...
}
```

Multiple people working on the same class

- The last and final real world use of partial classes is when more developers to work simultaneously in the same class.



Method Overloading



int abs (int i) ;
long abs (long l) ;
double abs (double d) ;

int f@absH (int i) ;
long f@absL (long l) ;
double f@absD (double d) ;

Number

Order

Type

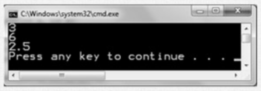
- Value
- Reference
- Output
- Parameter Array

Method Overloading

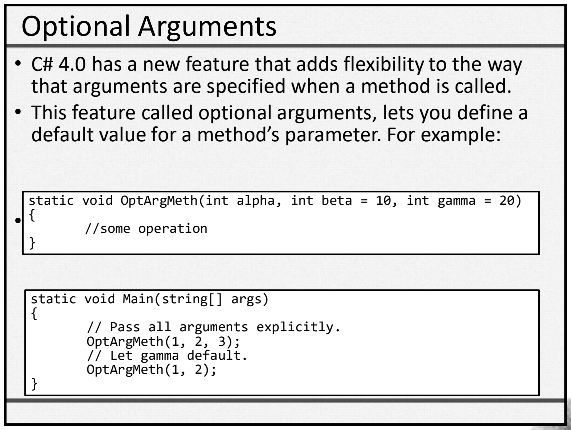
- Methods with the same name but different signature are called overloaded methods.
- Methods may differ in terms of
  - Number of parameters
  - Types of parameter
  - Order of parameter
- Note that if two methods are identical except for their return types or access specifiers, then the methods are not overloaded.
- Overloading works same for class and struct

Example

```
using System;
class Overload1
{
    static void add(int a, int b)
    {
        int c = 0;
        c = a + b;
        Console.WriteLine(c);
    }
    static void add(int a, int b, int c)
    {
        int x = 0;
        x = a + b + c;
        Console.WriteLine(x);
    }
    static void add(double a, double b)
    {
        double c = 0;
        c = a + b;
        Console.WriteLine(c);
    }
    public static void Main()
    {
        add(1, 2);
        add(1, 2, 3);
        add(1.2, 1.3);
    }
}
```



```
3
6
2.5
Press any key to continue . . .
```

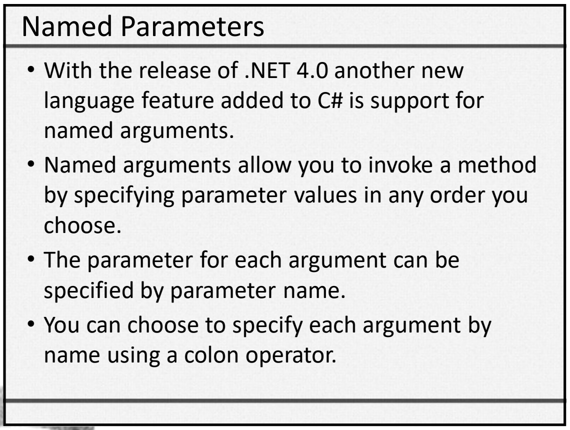


Optional Arguments

- C# 4.0 has a new feature that adds flexibility to the way that arguments are specified when a method is called.
- This feature called optional arguments, lets you define a default value for a method's parameter. For example:
-

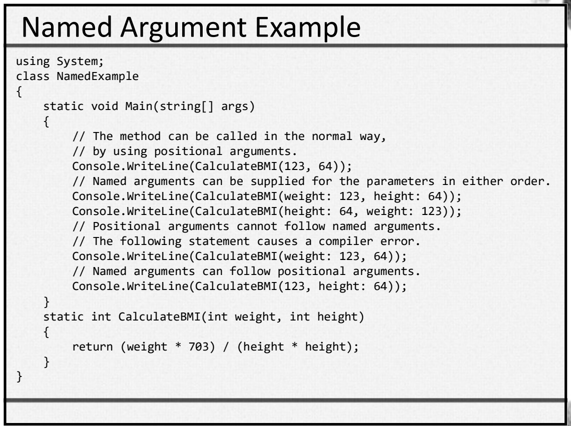
```
static void OptArgMeth(int alpha, int beta = 10, int gamma = 20)
{
        //some operation
}
```

```
static void Main(string[] args)
{
        // Pass all arguments explicitly.
        OptArgMeth(1, 2, 3);
        // Let gamma default.
        OptArgMeth(1, 2);
}
```



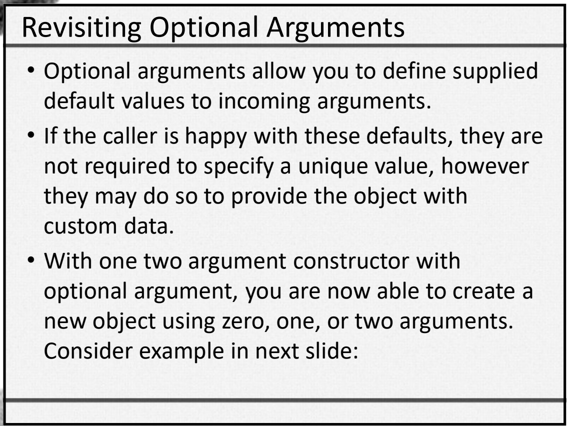
Named Parameters

- With the release of .NET 4.0 another new language feature added to C# is support for named arguments.
- Named arguments allow you to invoke a method by specifying parameter values in any order you choose.
- The parameter for each argument can be specified by parameter name.
- You can choose to specify each argument by name using a colon operator.



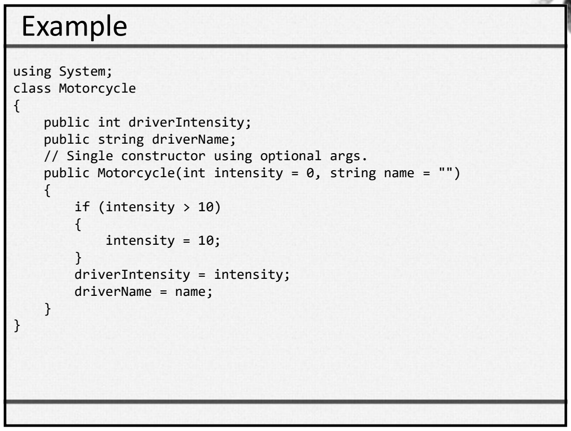
Named Argument Example

```
using System;
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way,
        // by using positional arguments.
        Console.WriteLine(CalculateBMI(123, 64));
        // Named arguments can be supplied for the parameters in either order.
        Console.WriteLine(CalculateBMI(weight: 123, height: 64));
        Console.WriteLine(CalculateBMI(height: 64, weight: 123));
        // Positional arguments cannot follow named arguments.
        // The following statement causes a compiler error.
        Console.WriteLine(CalculateBMI(weight: 123, 64));
        // Named arguments can follow positional arguments.
        Console.WriteLine(CalculateBMI(123, height: 64));
    }
    static int CalculateBMI(int weight, int height)
    {
        return (weight * 703) / (height * height);
    }
}
```



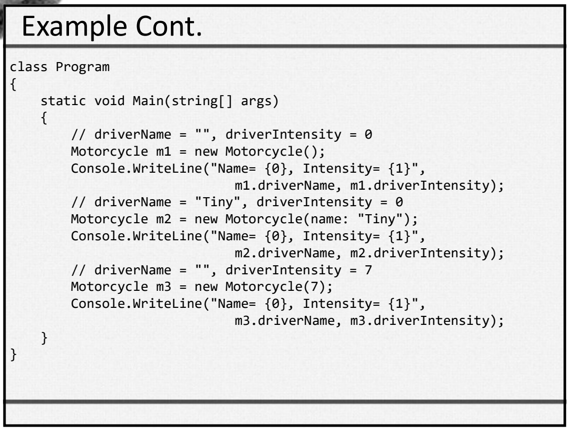
Revisiting Optional Arguments

- Optional arguments allow you to define supplied default values to incoming arguments.
- If the caller is happy with these defaults, they are not required to specify a unique value, however they may do so to provide the object with custom data.
- With one two argument constructor with optional argument, you are now able to create a new object using zero, one, or two arguments. Consider example in next slide:



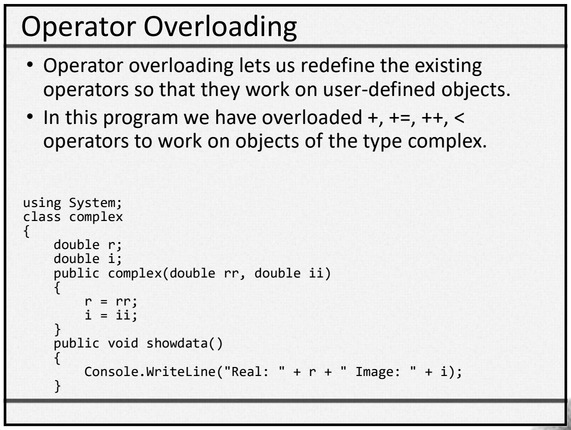
Example

```
using System;
class Motorcycle
{
    public int driverIntensity;
    public string driverName;
    // Single constructor using optional args.
    public Motorcycle(int intensity = 0, string name = "")
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
}
```



Example Cont.

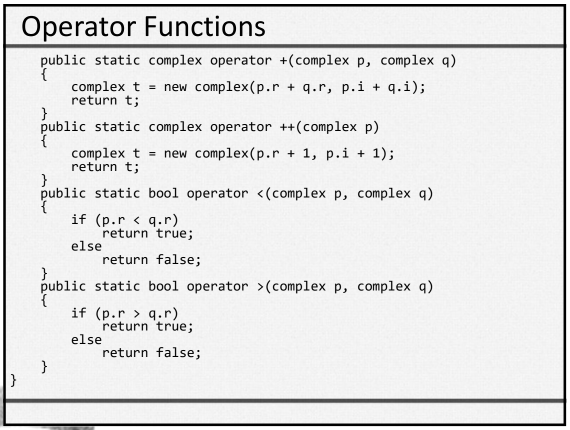
```
class Program
{
    static void Main(string[] args)
    {
        // driverName = "", driverIntensity = 0
        Motorcycle m1 = new Motorcycle();
        Console.WriteLine("Name= {0}, Intensity= {1}",
                             m1.driverName, m1.driverIntensity);
        // driverName = "Tiny", driverIntensity = 0
        Motorcycle m2 = new Motorcycle(name: "Tiny");
        Console.WriteLine("Name= {0}, Intensity= {1}",
                             m2.driverName, m2.driverIntensity);
        // driverName = "", driverIntensity = 7
        Motorcycle m3 = new Motorcycle(7);
        Console.WriteLine("Name= {0}, Intensity= {1}",
                             m3.driverName, m3.driverIntensity);
    }
}
```



Operator Overloading

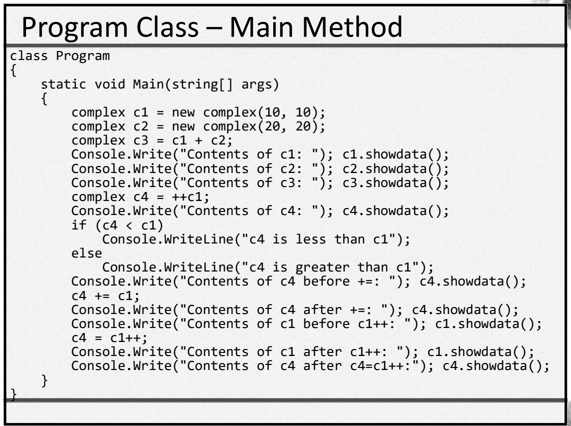
- Operator overloading lets us redefine the existing operators so that they work on user-defined objects.
- In this program we have overloaded +, +=, ++, < operators to work on objects of the type complex.

```
using System;
class complex
{
    double r;
    double i;
    public complex(double rr, double ii)
    {
        r = rr;
        i = ii;
    }
    public void showdata()
    {
        Console.WriteLine("Real: " + r + " Image: " + i);
    }
```



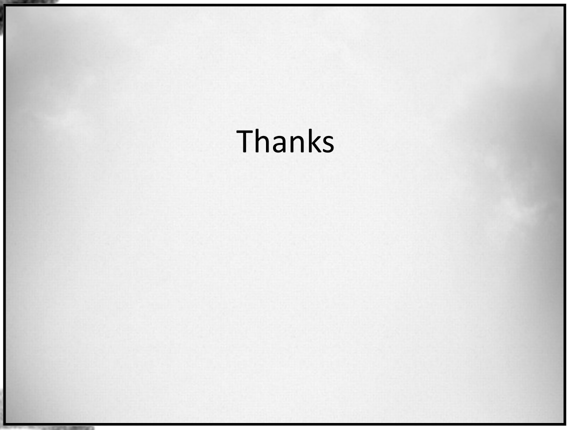
Operator Functions

```
    public static complex operator +(complex p, complex q)
    {
        complex t = new complex(p.r + q.r, p.i + q.i);
        return t;
    }
    public static complex operator ++(complex p)
    {
        complex t = new complex(p.r + 1, p.i + 1);
        return t;
    }
    public static bool operator <(complex p, complex q)
    {
        if (p.r < q.r)
            return true;
        else
            return false;
    }
    public static bool operator >(complex p, complex q)
    {
        if (p.r > q.r)
            return true;
        else
            return false;
    }
}
```



Program Class – Main Method

```
class Program
{
    static void Main(string[] args)
    {
        complex c1 = new complex(10, 10);
        complex c2 = new complex(20, 20);
        complex c3 = c1 + c2;
        Console.Write("Contents of c1: "); c1.showdata();
        Console.Write("Contents of c2: "); c2.showdata();
        Console.Write("Contents of c3: "); c3.showdata();
        complex c4 = ++c1;
        Console.Write("Contents of c4: "); c4.showdata();
        if (c4 < c1)
            Console.WriteLine("c4 is less than c1");
        else
            Console.WriteLine("c4 is greater than c1");
        Console.Write("Contents of c4 before +=: "); c4.showdata();
        c4 += c1;
        Console.Write("Contents of c4 after +=: "); c4.showdata();
        Console.Write("Contents of c1 before c1++: "); c1.showdata();
        c4 = c1++;
        Console.Write("Contents of c1 after c1++: "); c1.showdata();
        Console.Write("Contents of c4 after c4=c1++:"); c4.showdata();
    }
}
```



Thanks