

Agenda

- Defining Abstraction
- Levels of Abstraction
- Class as Abstraction
- Defining a C# Class
- Instantiating a Class and Accessing its Members
- Class Members
- Class Modifiers
- Member Modifiers
- Defining Encapsulation
- Encapsulating a Class
- Define Constructor
- Defining Properties



Defining Abstraction

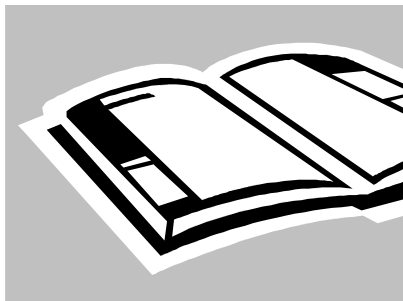
- An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer.
- Abstraction means ignoring the non-essential details of an object and concentrating on its essential details.

Different Levels of Abstraction

- Data Abstraction:
 - Constructs used to simplify the way information is presented to the programmer in programming language.
- Functional Abstraction:
 - Representation of very complex and low level instructions into more readable instructions.
- Object Abstraction:
 - Object Oriented Programming (OOP) languages take the concept even further and abstracts programming constructs as objects.

Everything is an Object

- An object is a structure that contains data and methods that manipulate data.
- All the objects in OOPs are based on the real world having a specific state and behavior.



Defining an Object

- In object-oriented programming, objects communicate and interact with each other using messages.

An object is a self-contained entity
with attributes and behaviors

Attributes

Information an object must know:

- Identity – uniqueness
- Attributes – structure
- State – current condition

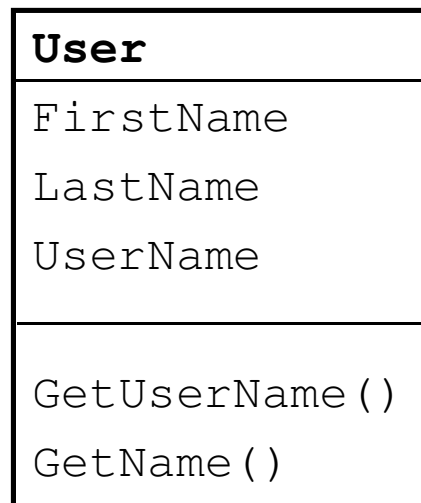
Behaviors

Behavior an object must do:

- Methods – what it can do
- Events – what it responds to

Class as Abstraction

- A class is an abstraction of its instances.
 - It defines all the attributes and methods that its instances must also have.



Defining a Class

- A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events.
- A class is like a blueprint.
- It defines the data and behavior of a type; objects are referred to as “instances” of a class.

Defining a C# Class

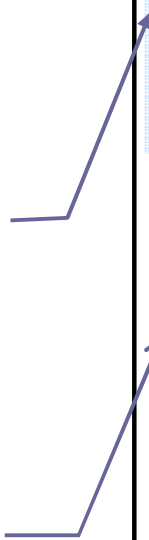
- A C# class denotes a category of objects, and acts as a *blueprint* for creating such objects.
- It defines its members referred to as *fields* and *methods*.
- The *fields* (also known as *variables* or *attributes*) refer to the properties of the class.
- The *methods* (also known as *operations*) refer to behaviors that the class exhibits.

```
using System;

class User
{
    string firstName;
    string lastName;
    string userName;
    string password;
    int Id;

    string GetUserName()
    {
        return userName;
    }

    string GetName()
    {
        return firstName + " " + lastName;
    }
}
```



Instantiating a Class & Accessing Its Members

- Instantiating a class means creating objects of its own type.
- The `new` operator is used to instantiate a class.

```
using System;
public class TestStatic
{
    public int length, breadth;
    public static int count;

    public int GetData(int p, int q)
    {
        int totaldata = p * q;
        return totaldata ;
    }

    public static int AddData(int m, int n)
    {
        int x = m * n;
        return x;
    }
}
```

```
class ImplementStatic
{
    public static void Main(string[] args)
    {
        // instantiating several objects
        TestStatic p1 = new TestStatic();
        TestStatic p2 = new TestStatic();

        // accessing instance variables
        p1.length = 4; p1.breadth = 4;
        // accessing static variables
        TestStatic.count = 2;

        // accessing instance variables
        int x = p1.length * p1.breadth;
        Console.WriteLine(" Accessing member variables =" + x);

        // accessing instance methods
        Console.WriteLine(" Accessing methods = " + p1.GetData(5,5));
        // accessing static method
        Console.WriteLine(" Accessing static methods= " + TestStatic.AddData(6,6));
    }
}
```

accessing instance variables

accessing static variables

accessing instance methods

accessing static methods

Class Members

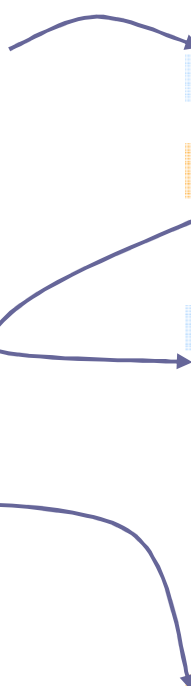
- A class member refers to one of the fields or methods of a class.
- *Instance members* are variables and methods belonging to objects where a copy of each variable and method is created for each object instantiated.
- *Static members* are variables and methods belonging to a class where only a single copy of variables and methods are shared by each object.
- *Constant* is a class member that represents a constant value: a value that can be computed at compile-time.

```
public class TestStatic
{
    //instance member variable
    public int length, breadth;

    //instance static member variable
    public static int count;

    //instance member method
    public int GetData(int p, int q)
    {
        int totalarea = p* q;
        return totalarea;
    }

    //static member method.
    public static int AddData(int m, int n)
    {
        int totalarea = m* n;
        return totalarea;
    }
}
```



Class Modifiers

- Modifiers change the way a class can be used.
- Access modifiers control the accessibility of a class.

Modifier	Description
<code>public</code>	Access is not limited.
<code>protected</code>	Accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.
<code>private</code>	Access limited to the containing type. By default access is private.
<code>internal</code>	Access limited to this program.
<code>sealed</code>	Class is prevented from inheriting.
<code>abstract</code>	Class that is incomplete and is intended to be used as base class.
<code>protected internal</code>	Available in the containing program or assembly and in the derived classes.

Member Modifiers (1 of 3)

Modifier	Description	Impact on Members
<code>new</code>	Allows you to declare a member with the same name or signature as an inherited member.	constant, field, property, event, method, indexer
<code>public</code>	Access is not limited.	field, method, property, event, indexer
<code>protected</code>	Access is limited to the containing class or types derived from the containing class.	field, method, property, event, indexer
<code>protected internal</code>	Member is available in the containing program or assembly and in the derived classes. <i>Includes both protected and internal modifiers.</i>	field, method, property, event, indexer
<code>internal</code>	Member is available within the assembly or component that is being created but not to the clients of that component.	field, method, property, event, indexer

Member Modifiers (2 of 3)

Modifier	Description	Impact on Members
<code>private</code>	Access is limited to the containing type. By default the access is private.	field, method, property, event, indexer
<code>abstract</code>	Is intended to be used as base class and the class itself is incomplete.	method, property, event, indexer
<code>static</code>	Does not operate on a specific instance of a member.	field, method, property, event, constant, indexer
<code>readonly</code>	Assignments to the fields can only occur in the declaration or in the same class constructor.	field
<code>volatile</code>	Field can be modified in the program by something like the operating system, the hardware, or a concurrently executing thread.	field

Member Modifiers (3 of 3)

Modifier	Description	Impact on Members
<code>sealed</code>	Specifies that a member cannot be inherited.	method, property, event, indexer
<code>extern</code>	Indicates that the method is implemented externally.	method, property, event, indexer
<code>virtual</code>	Declares a method or an accessor whose implementation can be changed by an overriding member in a derived class.	Method, property, event, indexer
<code>override</code>	Provides a new implementation of a virtual member inherited from a base class or provides an implementation of an abstract member inherited from an abstract class.	Method, property, event, indexer

Introduction to Constructors

- Basic initialization of an object can be done with constructor and is a special method, used when instantiating a class.
- Constructors are class members.
- Constructors should not have a return type.
- Constructors without parameters can be called default constructors.
- Constructors with parameters can be called parameter constructors.
- Constructors can be public, private, and static.

Introduction to Constructors (cont.)

Class User

```
{  
    int ID;  
    public User()  
    {  
        ID=0;  
        Console.WriteLine("Default Constructor called...");  
    }  
    public User(int Id)  
    {  
        ID=Id;  
        Console.WriteLine("Parameterised Constructor called...");  
    }  
}
```


Static Constructors

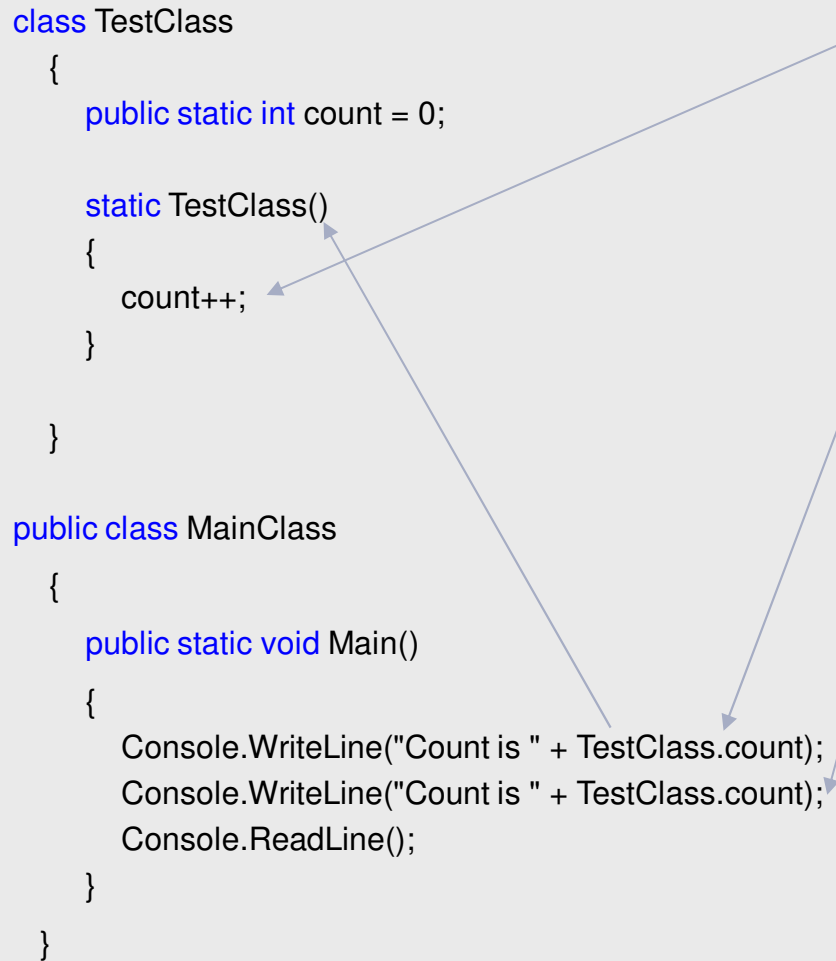
- Static constructors gets executed when static members of a class are accessed for the first time or when an object of the class is created.
- Static constructors have access to static members of their class only.
- A static constructor does not have parameters and access modifiers cannot be applied to it.
- Static variables get instantiated before the static constructor.
- Static constructors provide the flexibility of dynamically initializing static variables.

Static Constructors (cont.)

```
class TestClass
{
    public static int count = 0;

    static TestClass()
    {
        count++;
    }
}

public class MainClass
{
    public static void Main()
    {
        Console.WriteLine("Count is " + TestClass.count);
        Console.WriteLine("Count is " + TestClass.count);
        Console.ReadLine();
    }
}
```



Count is incremented to 1. Since static variables get instantiated before the static constructor execution, this operation is possible.

Static member count is being accessed for the first time. Hence the static constructor gets called.

Static member count is being accessed for the second time. Hence the static constructor will not be called. So count won't be incremented this time.

```
Count is 1
Count is 1
```

Defining Encapsulation

- Encapsulation provides the ability to hide internal details of an object from its users:
 - In programming, it is the process of combining elements to create a new entity:
 - For example: A procedure is a type of encapsulation because it combines a series of computer instructions.
 - A complex data type, such as a record or class, relies on encapsulation.
- Encapsulation is closely related to abstraction and information hiding.

Encapsulating a Class

- Members of a class must always be declared with the minimum level of visibility.
- Provide setters and getters (also known as accessor / mutators) to allow controlled access to private data.
- Provide other public methods (known as interfaces) that other objects must adhere to in order to interact with the object.

Properties

- A property is a member that provides access to a characteristic of an object or a class:
 - For example: Name of a customer, size of a font, caption of a window, etc.
- Properties are declared using property-declarations:
 - type property-name
- Property accessors specify the executable statements associated with reading and writing the property.
- The accessor-declarations enclosed in "{" and "}" tokens declare the accessors of the property.
- The accessor declarations consist of a get-accessor-declaration, a set-accessor-declaration, or both:
 - Each accessor declaration consists of the token get or set followed by an accessor-body.

```
private int number;

public int Number
{
    get //get accessor
    {
        return number;
    }
    set //set accessor
    {
        number = value;
    }
}
```

Encapsulation Example

```
using System;
class Loan
{
    // set variables to private
    private int id;

    //setting accessors
    public int ID
    {
        get
        {
            return id;
        }
        set
        {
            id = value;
        }
    }
}
```

```
static void Main(string[] args)
{
    // instantiate several objects
    Loan l1 = new Loan();
    Loan l2 = new Loan();
    Loan l3 = new Loan();

    // access instance variables using setters
    l1.ID=100;
    l2.ID=200;
    l3.ID=300;

    Console.WriteLine("Loan1 no is " + l1.ID);
    Console.WriteLine("Loan2 no is " + l2.ID);
    Console.WriteLine("Loan3 no is " + l3.ID);
}
```

```
Loan1 no is =100
Loan2 no is =200
Loan3 no is =300
```