Object Oriented Concepts in C#

Table of Contents

- Defining Classes
- Access Modifiers
- Constructors
- Fields, Constants and Properties
- Static Members
- Structures
- Delegates and Events
- Interfaces
- Inheritance
- Polymorphism

OOP and .NET

- In .NET Framework the object-oriented approach has roots in the deepest architectural level
- All .NET applications are object-oriented
- All .NET languages are object-oriented
- The class concept from OOP has two realizations:
  - Classes and structures
- There is no multiple inheritance in .NET
- Classes can implement several interfaces at the same time

Classes in OOP

- Classes model real-world objects and define
  - Attributes (state, properties, fields)
  - Behavior (methods, operations)
- Classes describe structure of objects
  - Objects describe particular instance of a class
- Properties hold information about the modeled object relevant to the problem
- Operations implement object behavior

Classes in C#

- Classes in C# could have following members:
  - Fields, constants, methods, properties, indexers, events, operators, constructors, destructors
  - Inner types (inner classes, structures, interfaces, delegates, ...)
- Members can have access modifiers (scope)
  - `public`, `private`, `protected`, `internal`
- Members can be
  - `static` (common) or specific for a given object

Simple Class Definition

```
public class Cat : Animal
{
   private string name;
   private string owner;

   public Cat(string name, string owner)
   {
      this.name = name;
      this.owner = owner;
   }

   public string Name
   {
      get { return name; }
      set { name = value; }
   }
```

Begin of class definition

Inherited (base) class

Fields

Constructor

Property

Simple Class Definition (2)

```
    public string Owner
    {
        get { return owner;}
        set { owner = value; }
    }

    public void SayMiau()
    {
        Console.WriteLine("Miauuuuuuu!");
    }
}
```

Method

End of class definition

Class Definition and Members

- Class definition consists of:
  - Class declaration
  - Inherited class or implemented interfaces
  - Fields (static or not)
  - Constructors (static or not)
  - Properties (static or not)
  - Methods (static or not)
  - Events, inner types, etc.

Access Modifiers

Public, Private, Protected, Internal

Access Modifiers

- Class members can have access modifiers
  - Used to restrict the classes able to access them
  - Supports the OOP principle "encapsulation"
- Class members can be:
  - `public` – accessible from any class
  - `protected` – accessible from the class itself and all its descendent classes
  - `private` – accessible from the class itself only
  - `internal` – accessible from the current assembly (used by default)

Defining Classes

Example

Task: Define Class Dog

- Our task is to define a simple class that represents information about a dog
  - The dog should have name and breed
  - If there is no name or breed assigned to the dog, it should be named "Balkan" and its breed should be "Street excellent"
  - It should be able to view and change the name and the breed of the dog
  - The dog should be able to bark

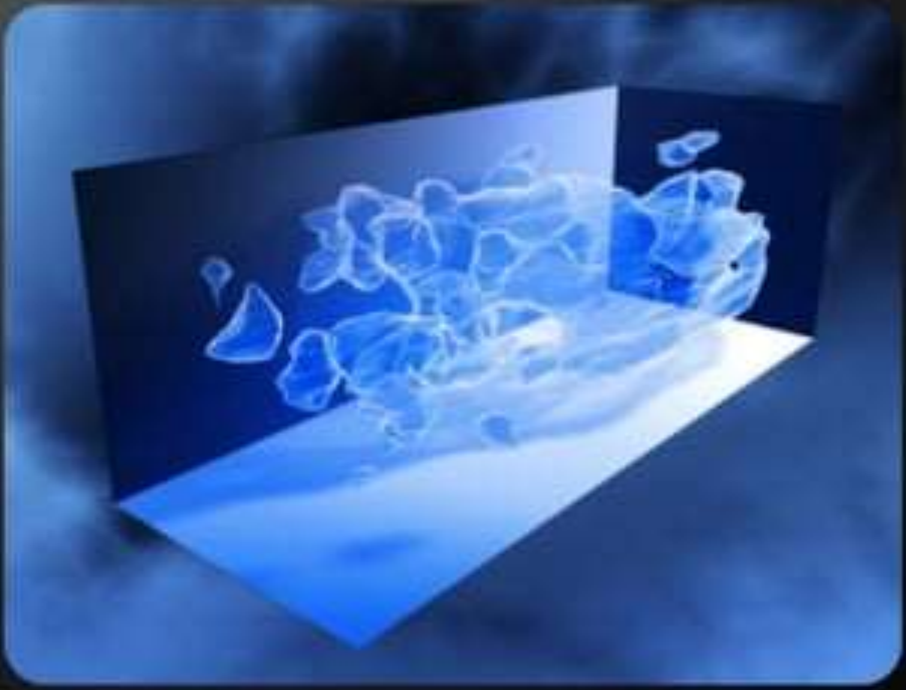
Defining Class Dog – Example

```
public class Dog
{
    private string name;
    private string breed;

    public Dog()
    {
        this.name = "Balkan";
        this.breed = "Street excellent";
    }

    public Dog(string name, string breed)
    {
        this.name = name;
        this.breed = breed;
    }
                                    //(example
continues)
```

Defining Class Dog – Example (2)

```
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string Breed
    {
        get { return breed; }
        set { breed = value; }
    }

    public void SayBau()
    {
        Console.WriteLine("{0} said: Bauuuuuu!",
name);
    }
}
```

Using Classes and Objects



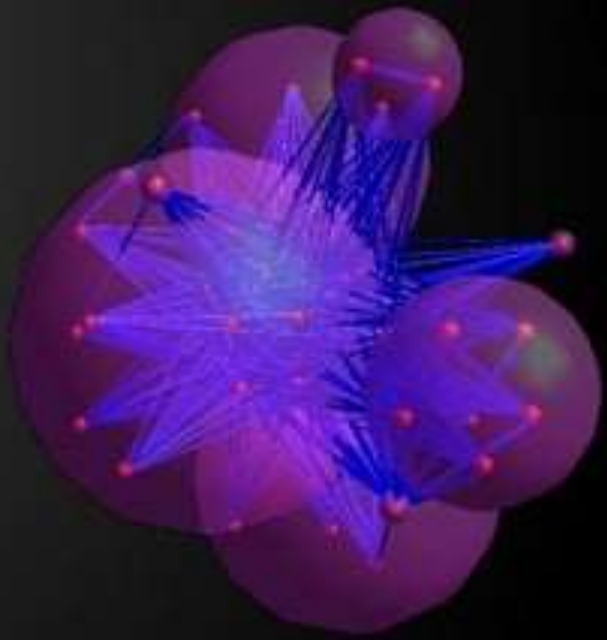
Using Classes

- How to use classes?
  - Create a new instance
  - Access the properties of the class
  - Invoke methods
  - Handle events

- How to define classes?
  - Create new class and define its members
  - Create new class using some other as base class

How to Use Classes (Non-static)?

- Create an instance
  - Initialize fields
- Manipulate instance
  - Read / change properties
  - Invoke methods
  - Handle events
- Release occupied resources
  - Done automatically in most cases



Task: Dog Meeting

- Our task is as follows:
  - Create 3 dogs
    - First should be named "Sharo", second – "Rex" and the last – left without name
  - Add all dogs in an array
  - Iterate through the array elements and ask each dog to bark
  - Note:
    - Use the Dog class from the previous example!

Dog Meeting – Example

```
static void Main()
{
   Console.WriteLine("Enter first dog's name: ");
   dogName = Console.ReadLine();
   Console.WriteLine("Enter first dog's breed: ");
   dogBreed = Console.ReadLine();

   // Using the Dog constructor to set name and breed
   Dog firstDog = new Dog(dogName, dogBreed);
   Dog secondDog = new Dog();
   Console.WriteLine("Enter second dog's name: ");
   dogName = Console.ReadLine();
   Console.WriteLine("Enter second dog's breed: ");
   dogBreed = Console.ReadLine();

   // Using properties to set name and breed
   secondDog.Name = dogName;
   secondDog.Breed = dogBreed;
}
```

Constructors

Defining and Using Class Constructors

What is Constructor?

- Constructors are special methods
  - Invoked when creating a new instance of an object
  - Used to initialize the fields of the instance
- Constructors has the same name as the class
  - Have no return type
  - Can have parameters
  - Can be `private`, `protected`, `internal`, `public`

Defining Constructors

- Class Point with parameterless constructor:

```
public class Point
{
    private int xCoord;
    private int yCoord;

    // Simple default constructor
    public Point()
    {
        xCoord = 0;
        yCoord = 0;
    }

    // More code ...
}
```

Defining Constructors (2)

```
public class Person
{
    private string name;
    private int age;
    // Default constructor
    public Person()
    {
        name = "[no name]";
        age = 0;
    }
    // Constructor with parameters
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    // More code ...
}
```

As rule constructors should initialize all own class fields.

Constructors and Initialization

- Pay attention when using inline initialization!

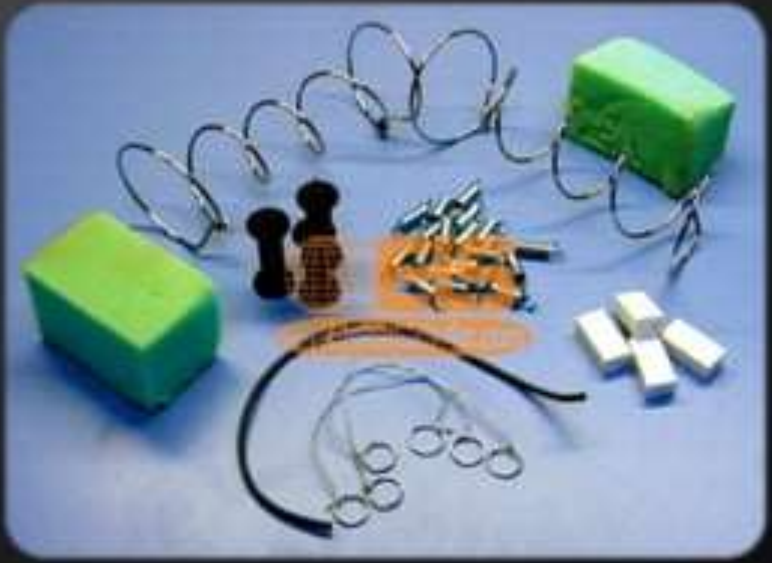
```
public class ClockAlarm
{
   private int hours = 9; // Inline initialization
   private int minutes = 0; // Inline initialization
   // Default constructor
   public ClockAlarm()
   { }
   // Constructor with parameters
   public ClockAlarm(int hours, int minutes)
   {
      this.hours = hours;      // Invoked after the
  inline
      this.minutes = minutes;  // initialization!
   }
   // More code ...
}
```

Chaining Constructors Calls

- Reusing constructors

```
public class Point
{
    private int xCoord;
    private int yCoord;

    public Point() : this(0,0) // Reuse constr
    {
    }

    public Point(int xCoord, int yCoord)
    {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }

    // More code ...
}
```

Fields, Constants and and Properties



Fields

- Fields contain data for the class instance
- Can be arbitrary type
- Have given scope
- Can be declared with a specific value

```
class Student
{
    private string firstName;
    private string lastName;
    private int course = 1;
    private string speciality;
    protected Course[] coursesTaken;
    private string remarks = "(no remarks)";
}
```

Constants

- Constant fields are defined like fields, but:
  - Defined with const
  - Must be initialized at their definition
  - Their value can not be changed at runtime

```
public class MathConstants
{
    public const string PI_SYMBOL = "π";
    public const double PI = 3.14159265358979323;
    public const double E = 2.718281828459045;
    public const double LN10 = 2.302585092994046;
    public const double LN2 = 0.693147180559945;
}
```

Read-Only Fields

- Initialized at the definition or in the constructor
  - Can not be modified further
- Defined with the keyword `readonly`
- Represent runtime constants

```
public class ReadOnlyDemo
{
    private readonly int size;
    public ReadOnlyDemo(int Size)
    {
      size = Size; // can not be further modified!
    }
}
```

The Role of Properties

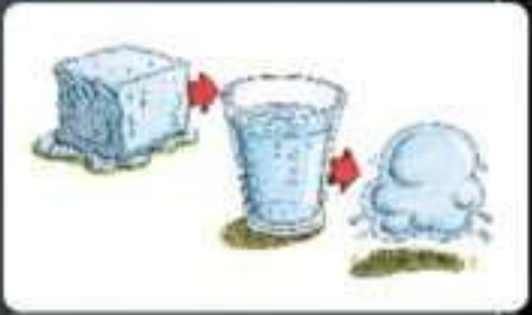
- Expose object's data to the outside world
- Control how the data is manipulated
- Properties can be:
  - Read-only
  - Write-only
  - Read and write
- Give good level of abstraction
- Make writing code easier

Defining Properties in C#

- Properties should have:
  - Access modifier (`public`, `protected`, etc.)
  - Return type
  - Unique name
  - Get and / or Set part
  - Can contain code processing data in specific way

Defining Properties – Example

```
public class Point
{
    private int xCoord;
    private int yCoord;

    public int XCoord
    {
        get { return xCoord; }
        set { xCoord = value; }
    }

    public int YCoord
    {
        get { return yCoord; }
        set { yCoord = value; }
    }

    // More code ...
}
```



Dynamic Properties

- Properties are not obligatory bound to a class field – can be calculated dynamically:

```
public class Rectangle
{
    private float width;
    private float height;

    // More code ...

    public float Area
    {
        get
        {
            return width * height;
        }
    }
}
```

Automatic Properties

- Properties could be defined without an underlying field behind them
  - It is automatically created by the C# compiler

```
class UserProfile
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
…
UserProfile profile = new UserProfile() {
    FirstName = "Steve",
    LastName = "Balmer",
    UserId = 91112 };
```

Static Members

Static vs. Instance Members

Static Members

- Static members are associated with a type rather than with an instance
  - Defined with the modifier `static`
- Static can be used for
  - Fields
  - Properties
  - Methods
  - Events
  - Constructors

Static vs. Non-Static

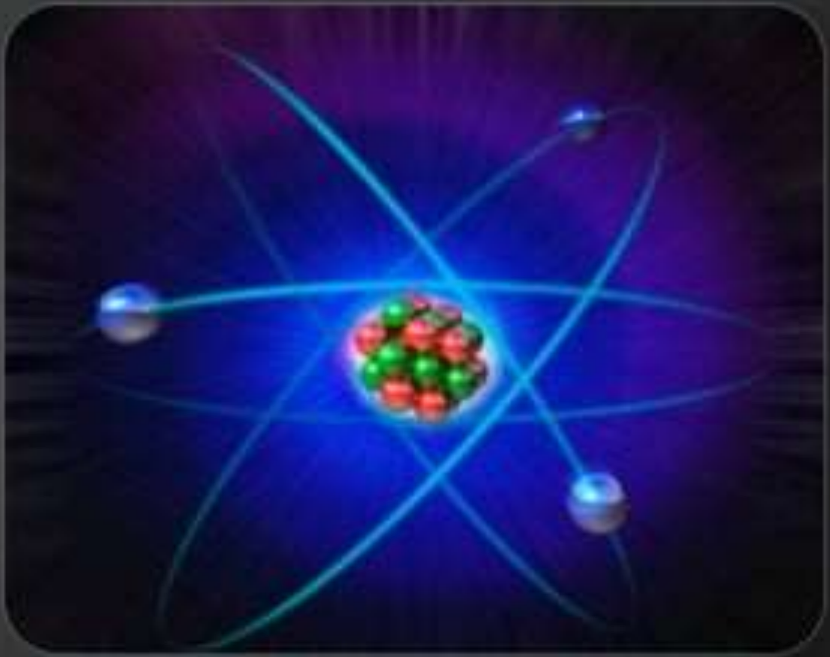
- Static:
  - Associated with a type, not with an instance
- Non-Static:
  - The opposite, associated with an instance
- Static:
  - Initialized just before the type is used for the first time
- Non-Static:
  - Initialized when the constructor is called

Static Members – Example

```
public class SqrtPrecalculated
{
    public const int MAX_VALUE = 10000;

    // Static field
    private static int[] sqrtValues;

    // Static constructor
    private static SqrtPrecalculated()
    {
        sqrtValues = new int[MAX_VALUE + 1];
        for (int i = 0; i < sqrtValues.Length; i++)
        {
            sqrtValues[i] = (int)Math.Sqrt(i);
        }
    }
                                        //(example continues)
```

Static Members – Example (2)

```
   // Static method
   public static int GetSqrt(int value)
   {
      return sqrtValues[value];
   }

   // The Main() method is always static
   static void Main()
   {
      Console.WriteLine(GetSqrt(254));
   }
}
```



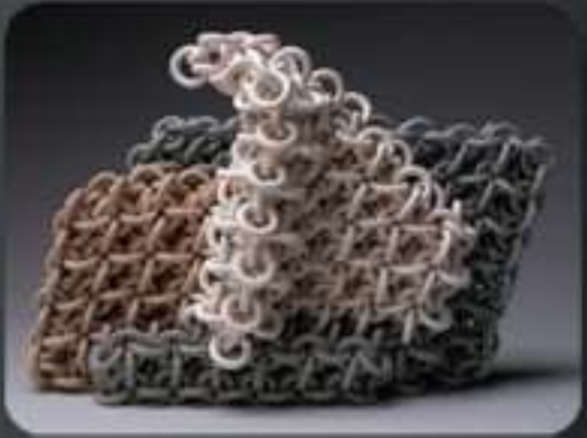
Structures

Structures

- Structures represent a combination of fields with data
  - Look like the classes, but are value types
  - Their content is stored in the stack
  - Transmitted by value
  - Destroyed when go out of scope
- However classes are reference type and are placed in the dynamic memory (heap)
  - Their creation and destruction is slower

Structures – Example

```
struct Point
{
    public int X, Y;
}

struct Color
{
    public byte redValue;
    public byte greenValue;
    public byte blueValue;
}

struct Square
{
    public Point location;
    public int size;
    public Color borderColor;
    public Color surfaceColor;
}
```



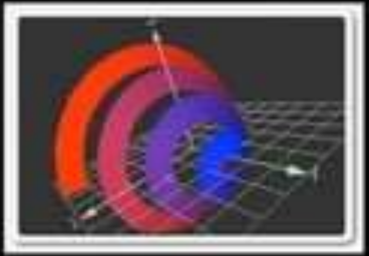
When to Use Structures?

- Use structures
  - To make your type behave as a primitive type
  - If you create many instances and after that you free them – e.g. in a cycle
- Do not use structures
  - When you often transmit your instances as method parameters
  - If you use collections without generics (too much boxing / unboxing!)

Delegates and Events

What are Delegates?

- Delegates are reference types
- Describe the signature of a given method
  - Number and types of the parameters
  - The return type
- Their "values" are methods
  - These methods correspond to the signature of the delegate



What are Delegates? (2)

- Delegates are roughly similar to function pointers in C and C++
  - Contain a strongly-typed pointer (reference) to a method
- They can point to both static or instance methods
- Used to perform callbacks

Delegates – Example

```
// Declaration of a delegate
public delegate void SimpleDelegate(string param);

public class TestDelegate
{
   public static void TestFunction(string param)
   {
      Console.WriteLine("I was called by a delegate.");
      Console.WriteLine("I got parameter {0}.", param);
   }
   public static void Main()
   {
      // Instantiation of a delegate
      SimpleDelegate simpleDelegate =
       new SimpleDelegate(TestFunction);
      // Invocation of the method, pointed by a
  delegate
      simpleDelegate("test");
    }
}
```

telerik

Anonymous Methods

- We are sometimes forced to create a class or a method just for the sake of using a delegate
  - The code involved is often relatively short and simple
- Anonymous methods let you define an nameless method called by a delegate
  - Less coding
  - Improved code readability

Using Delegates: Standard Way

```
class SomeClass
{
    delegate void SomeDelegate(string str);

    public void InvokeMethod()
    {
        SomeDelegate dlg = new
  SomeDelegate(SomeMethod);
        dlg("Hello");
    }

    void SomeMethod(string str)
    {
        Console.WriteLine(str);
    }
}
```

Using Anonymous Methods

- The same thing can be accomplished by using an anonymous method:

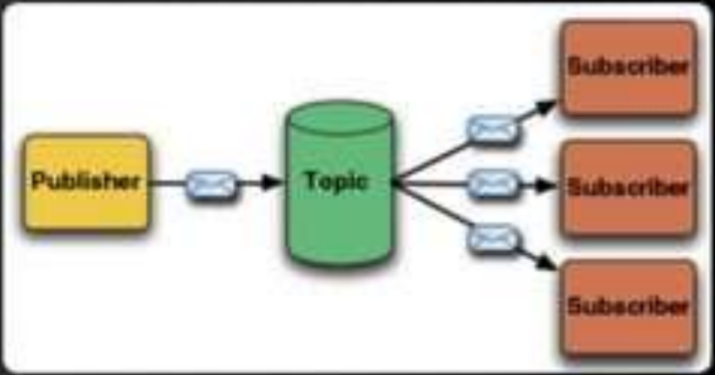
```
class SomeClass
{
    delegate void SomeDelegate(string str);

    public void InvokeMethod()
    {
        SomeDelegate dlg = delegate(string str)
        {
            Console.WriteLine(str);
        };
        dlg("Hello");
    }
}
```

Events

- In component-oriented programming the components send events to their owner to notify them when something happens
  - E.g. when a button is pressed an event is raised
- The object which causes an event is called event sender
- The object which receives an event is called event receiver
- In order to be able to receive an event the event receivers must first "subscribe for the event"

Events in .NET

- In the component model of .NET Framework delegates and events provide mechanism for:
  - Subscription to an event
  - Sending an event
  - Receiving an event



- Events in C# are special instances of delegates declared by the C# keyword event
  - Example (`Button.Click`):

```
public event EventHandler Click;
```

Events in .NET (2)

- The C# compiler automatically defines the += and -= operators for events
  - += subscribe for an event
  - -= unsubscribe for an event
- There are no other allowed operations
- Example:

```
Button button = new Button("OK");
button.Click += delegate
{
    Console.WriteLine("Button clicked.");
};
```

Events vs. Delegates

- Events are not the same as member fields of type delegate

```
public MyDelegate m;   ≠   public event MyDelegate m;
```

- The event is processed by a delegate
- Events can be members of an interface unlike delegates
- Calling of an event can only be done in the class it is defined in
- By default the access to the events is synchronized (thread-safe)

System.EventHandler Delegate

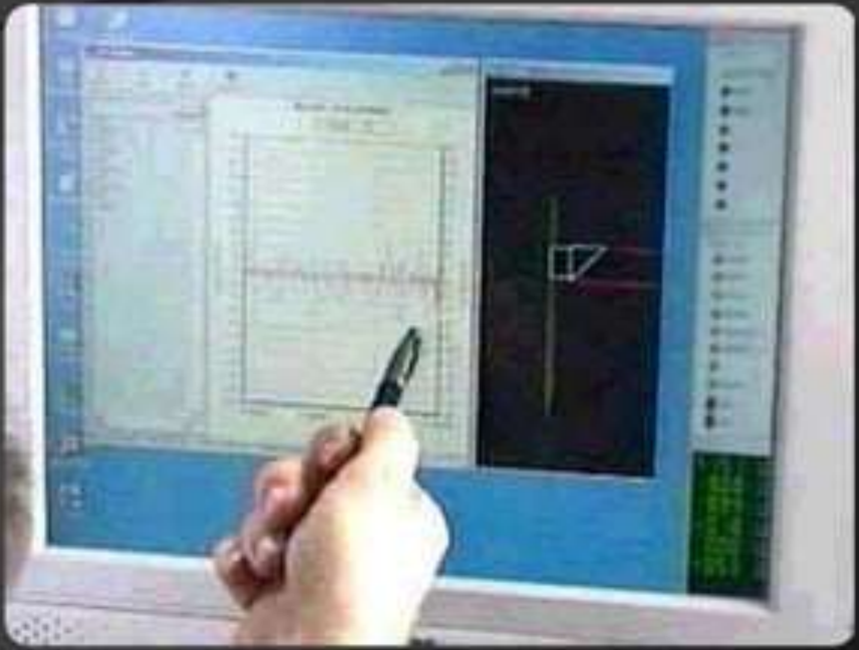
- Defines a reference to a callback method, which handles events
  - No additional information is sent

```
public delegate void EventHandler(
    Object sender, EventArgs e);
```

- Used in many occasions internally in .NET
  - E.g. in ASP.NET and Windows Forms
- The EventArgs class is base class with no information about the event
  - Sometimes delegates derive from it

EventHandler – Example

```
public class Button
{
   public event EventHandler Click;
   public event EventHandler GotFocus;
   public event EventHandler TextChanged;
   ...
}
public class ButtonTest
{
   private static void Button_Click(object sender,
    EventArgs eventArgs)
   {
      Console.WriteLine("Call Button_Click() event");
   }
   public static void Main()
   {
      Button button = new Button();
      button.Click += Button_Click;
   }
}
```



Interfaces and Abstract Classes

Interfaces

- Describe a group of methods (operations), properties and events
  - Can be implemented by given `class` or `structure`
- Define only the methods' prototypes
- No concrete implementation
- Can be used to define `abstract` data types
- Can not be instantiated
  - Members do not have scope modifier and by default the scope is `public`

Interfaces – Example

```
public interface IPerson
{
    string Name  // property Name
    { get; set; }
    DateTime DateOfBirth  // property Dat
    { get; set; }
    int Age  // property Age (read-only)
    { get; }
}
```

Interfaces – Example (2)

```
interface IShape
{
    void SetPosition(int x, int y);
    int CalculateSurface();
}

interface IMovable
{
    void Move(int deltaX, int deltaY);
}

interface IResizable
{
    void Resize(int weight);
    void Resize(int weightX, int weightY);
    void ResizeByX(int weightX);
    void ResizeByY(int weightY);
}
```

Interface Implementation

- Classes and structures can implement (support) one or many interfaces
- Interface realization must implement all its methods
- If some methods do not have implementation the `class` or `structure` have to be declared as an abstract

Interface Implementation – Example

```
class Rectangle : IShape, IMovable
{
   private int x, y, width, height;
   public void SetPosition(int x, int y) // IShape
   {
      this.x = x;
      this.y = y;
   }
   public int CalculateSurface() // IShape
   {
      return this.width * this.height;
   }
   public void Move(int deltaX, int deltaY) // IMovable
   {
      this.x += deltaX;
      this.y += deltaY;
   }
}
```

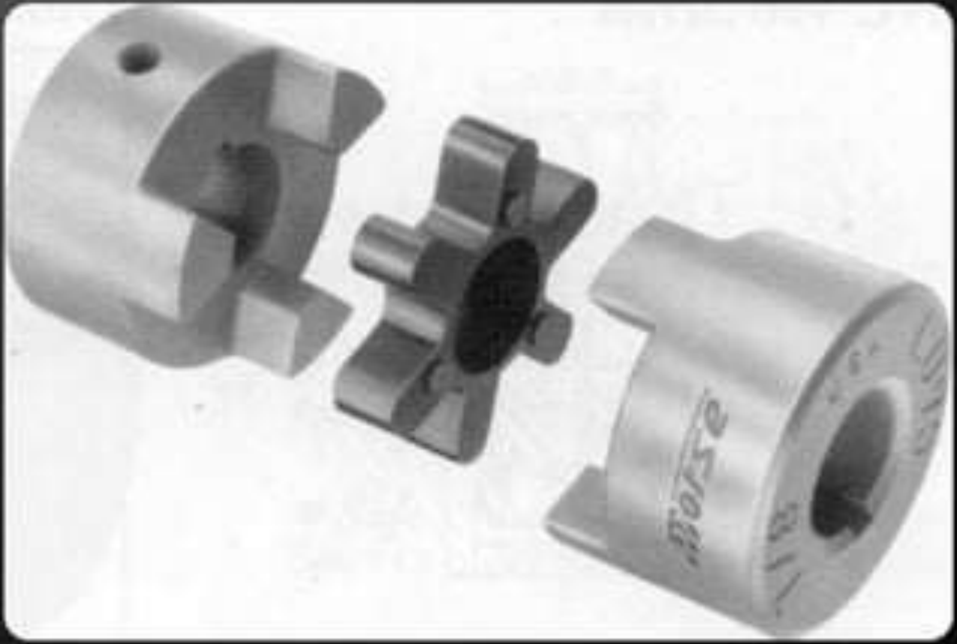
Abstract Classes

- Abstract method is a method without implementation
  - Left empty to be implemented by descendant classes
- When a class contains at least one abstract method, it is called abstract class
  - Mix between class and interface
  - Inheritors are obligated to implement their abstract methods
  - Can not be directly instantiated

Abstract Class – Example

```
abstract class MovableShape : IShape, IMovable
{
   private int x, y;
   public void Move(int deltaX, int deltaY)
   {
      this.x += deltaX;
      this.y += deltaY;
   }
   public void SetPosition(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   public abstract int CalculateSurface();
}
```

Cohesion and Coupling



Cohesion

- Cohesion describes how closely all the routines in a class or all the code in a routine support a central purpose
  - Cohesion must be strong
  - Classes must contain strongly related functionality and aim for single purpose
  - Cohesion is a useful tool for managing complexity
  - Well-defined abstractions keep cohesion strong

Good and Bad Cohesion

- Good cohesion: hard disk, CD-ROM, floppy

- BAD: spaghetti code

Strong Cohesion

- Strong cohesion example
  - Class Math that has methods:
    - Sin(), Cos(), Asin(), Sqrt(), Pow(), Exp()
    - Math.PI, Math.E

```
double sideA = 40, sideB = 69;
double angleAB = Math.PI / 3;

double sideC =
    Math.Pow(sideA, 2) + Math.Pow(sideB, 2)

    - 2 * sideA * sideB * Math.Cos(angleAB);

double sidesSqrtSum = Math.Sqrt(sideA) +
  Math.Sqrt(sideB) + Math.Sqrt(sideC);
```

Bad Cohesion

- Example of bad cohesion
- Class Magic that has all these methods:

```
public void PrintDocument(Document d);

public void SendEmail(string recipient, string
  subject, string text);

public void CalculateDistanceBetweenPoints(int x1,
  int y1, int x2, int y2)
```

- Another example:

```
MagicClass.MakePizza("Fat Pepperoni");

MagicClass.WithdrawMoney("999e6");

MagicClass.OpenDBConnection();
```

Coupling

- Coupling describes how tightly a class or routine is related to other classes or routines
- Coupling must be kept loose
  - Modules must depend little on each other
  - All classes and routines must have small, direct, visible, and flexible relations to other classes and routines
  - One module must be easily used by other modules

Loose and Tight Coupling

- Loose Coupling:
  - Easily replace old HDD
  - Easily place this HDD to another motherboard

- Tight Coupling:
  - Where is the video adapter?
  - Can you change the video controller?

Loose Coupling – Example

```
class Report
{
    public bool LoadFromFile(string fileName) {…}
    public bool SaveToFile(string fileName) {…}
}

class Printer
{
    public static int Print(Report report) {…}
}

class LooseCouplingExample
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile("C:\\DailyReport.rep");
        Printer.Print(myReport);
    }
}
```

Tight Coupling – Example

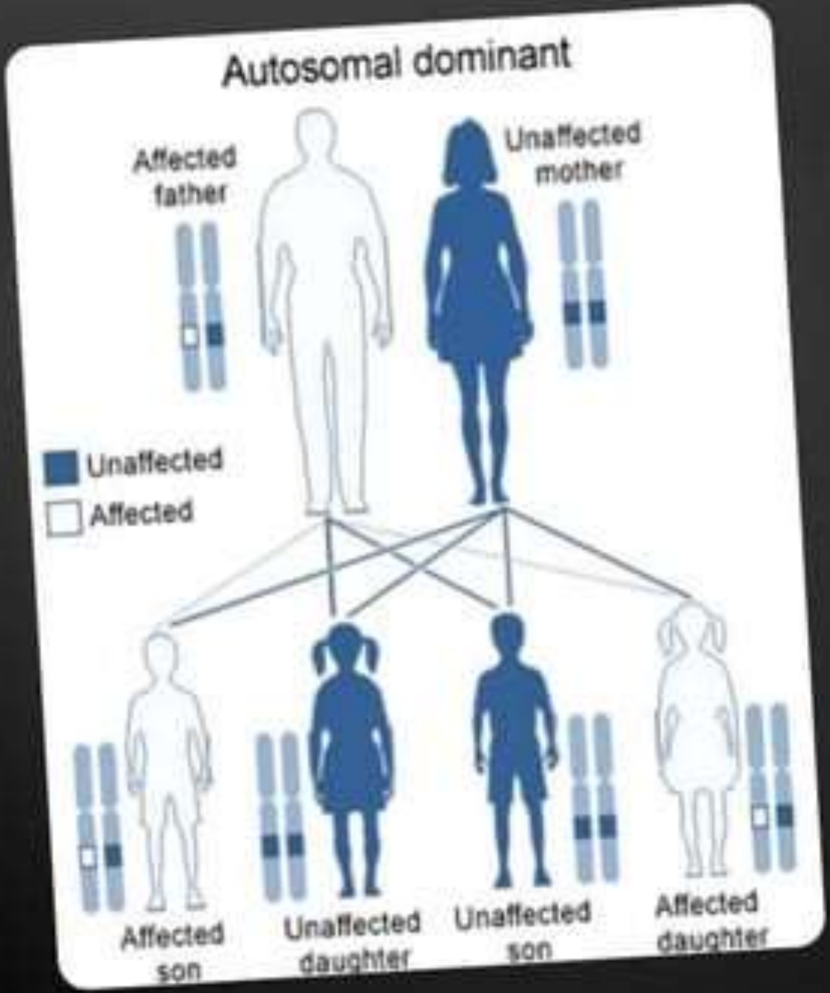
```
class MathParams
{
    public static double operand;
    public static double result;
}
class MathUtil
{
    public static void Sqrt()
    {
      MathParams.result =
  CalcSqrt(MathParams.operand);
    }
}
class Example
{
    static void Main()
    {
        MathParams.operand = 64;
        MathUtil.Sqrt();
        Console.WriteLine(MathParams.result);
    }
}
```

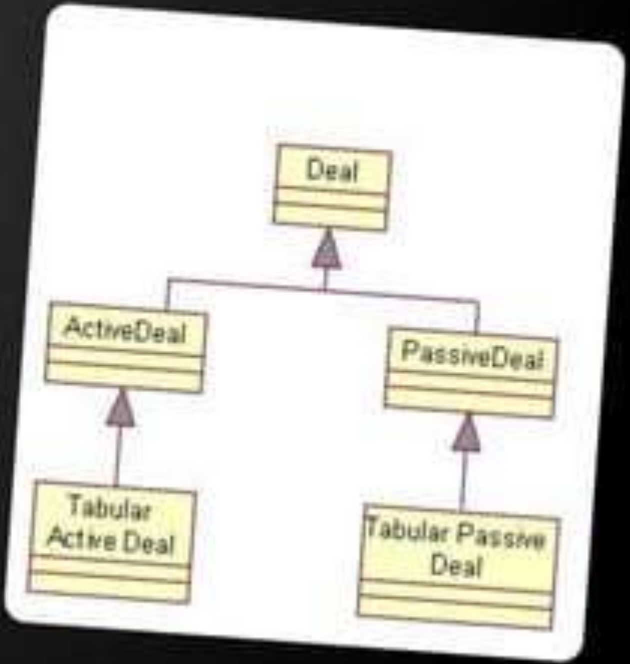
Spaghetti Code

- Combination of bad cohesion and tight coupling

```
class Report
{
    public void Print() {…}
    public void InitPrinter() {…}
    public void LoadPrinterDriver(string fileName) {…}
    public bool SaveReport(string fileName) {…}
    public void SetPrinter(string printer) {…}
}

class Printer
{
    public void SetFileName() {…}
    public static bool LoadReport() {…}
    public static bool CheckReport() {…}
}
```

Inheritance





Inheritance

- Inheritance is the ability of a class to implicitly gain all members from another class
  - Inheritance is fundamental concept in OOP
- The class whose methods are inherited is called base (parent) class
- The class that gains new functionality is called derived (child) class
- Inheritance establishes an is-a relationship between classes: A is B

Inheritance (2)

- All class members are inherited
  - Fields, methods, properties, …
- In C# classes could be inherited
  - The structures in C# could not be inherited
- Inheritance allows creating deep inheritance hierarchies
- In .NET there is no multiple inheritance, except when implementing interfaces

How to Define Inheritance?

- We must specify the name of the base class after the name of the derived

```
public class Shape
{...}
public class Circle : Shape
{...}
```

- In the constructor of the derived class we use the keyword base to invoke the constructor of the base class

```
public Circle (int x, int y) : base(x)
{...}
```

Inheritance – Example

```
public class Mammal
{
   private int age;

   public Mammal(int age)
   {
      this.age = age;
   }

   public int Age
   {
      get { return age; }
      set { age = value; }
   }

   public void Sleep()
   {
      Console.WriteLine("Shhh! I'm sleeping!");
   }
}
```

Inheritance – Example (2)

```
public class Dog : Mammal
{
   private string breed;
   public Dog(int age, string breed): base(age)
   {
      this.breed = breed;
   }
   public string Breed
   {
      get { return breed; }
      set { breed = value; }
   }
   public void WagTail()
   {
      Console.WriteLine("Tail wagging...");
   }
}
```

Inheritance – Example (3)

```
static void Main()
{
    // Create 5 years old mammal
    Mamal mamal = new Mamal(5);
    Console.WriteLine(mamal.Age);
    mamal.Sleep();

    // Create a bulldog, 3 years old
    Dog dog = new Dog("Bulldog", 3);
    dog.Sleep();
    dog.Age = 4;
    Console.WriteLine("Age: {0}", dog.Age);
    Console.WriteLine("Breed: {0}", dog.Breed);
    dog.WagTail();
}
```

Polymorphism

Polymorphism

- Polymorphism is fundamental concept in OOP
  - The ability to handle the objects of a specific class as instances of its parent class and to call abstract functionality
- Polymorphism allows creating hierarchies with more valuable logical structure
  - Allows invoking abstract functionality without caring how and where it is implemented

Polymorphism (2)

- Polymorphism is usually implemented through:
  - Virtual methods (`virtual`)
  - Abstract methods (`abstract`)
  - Methods from an interface (`interface`)
- In C# to override virtual method the keyword `override` is used
- C# allows hiding virtual methods in derived classes by the keyword `new`

Polymorphism – Example

```
class Person
{
    public virtual void PrintName()
    {
        Console.WriteLine("I am a person."
    }
}
class Trainer : Person
{
    public override void PrintName()
    {
        Console.WriteLine("I am a trainer.");
    }
}
class Student : Person
{
    public override void PrintName()
    {
        Console.WriteLine("I am a student.");
    }
}
```

Polymorphism – Example (2)

```
static void Main()
{
    Person[] persons =
    {
        new Person(),
        new Trainer(),
        new Student()
    };
    foreach (Person p in persons)
    {
        Console.WriteLine(p);
    }

    // I am a person.
    // I am a trainer.
    // I am a student.
}
```

Questions?