Object-Oriented Programming with C#

UIC School of Engineering – MIAGE 2

Table of Contents

OOP and .NET

- In .NET Framework the object-oriented approach has roots at 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
  - Still classes can implement several interfaces at the same time

Defining Classes

What is Class?

- The formal definition of class:

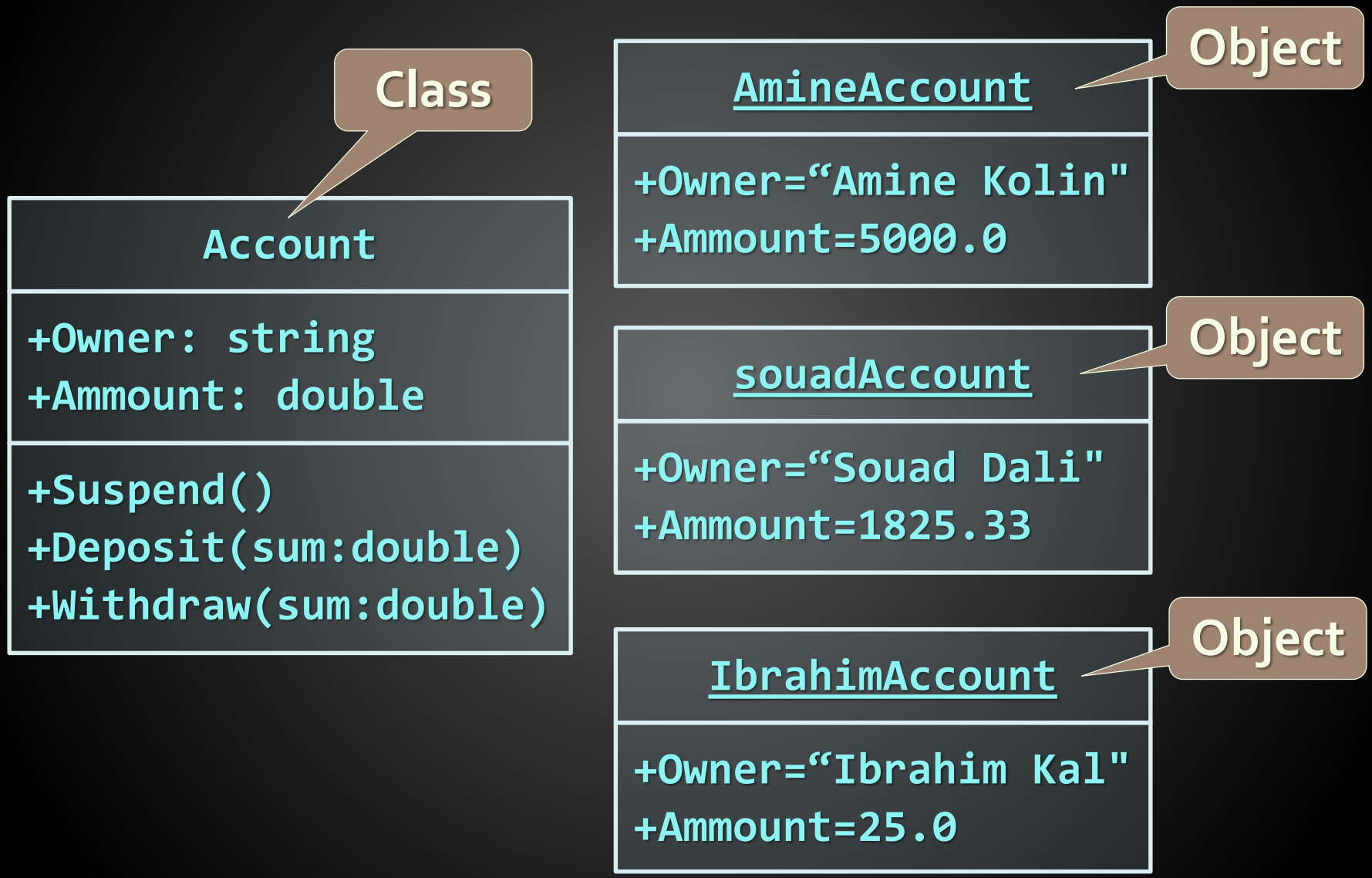
> Classes act as templates from which an instance of an object is created at run time. Classes define the properties of the object and the methods used to control the object's behavior.

- Classes define:
  - Set of attributes Represented by fields and properties
  - Set of actions (behavior) Represented by methods

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

Objects – Example



Simple Class Definition

Begin of 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; }
    }
```

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

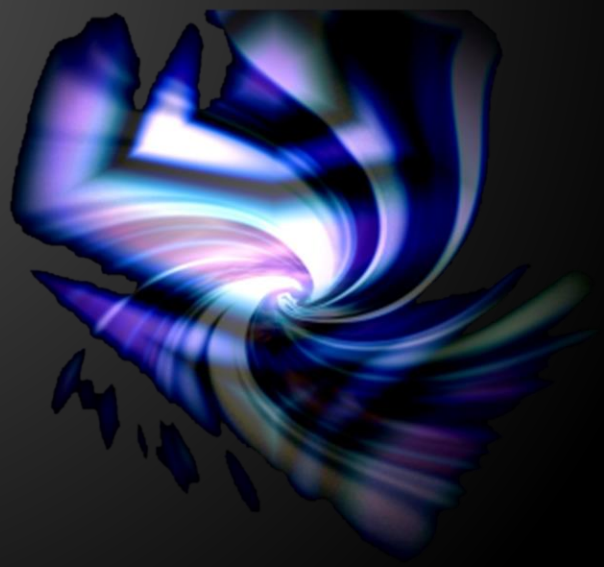
End of class definition

Classes and Their Members

- Classes have members
  - Fields, constants, methods, properties, indexers, events, operators, constructors, destructors
  - Inner types (inner classes, structures, interfaces, delegates, …)
- Members have modifiers (scope)
  - public, private, protected, internal, protected internal
- Members can be
  - static (common) or for a given type

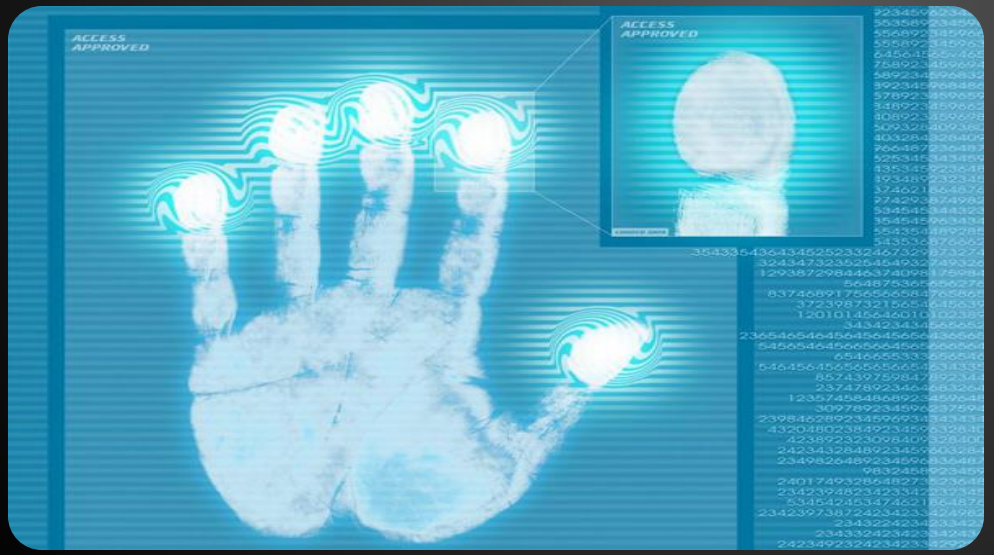
Class Definition and Members

- 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, Protected Internal



Access Modifiers

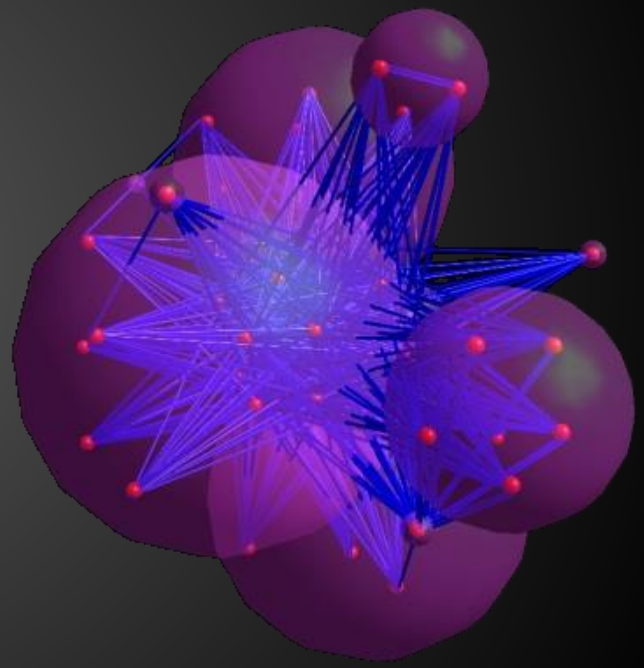
- Class members can have access modifiers
  - Used to restrict the classes able to access them
  - Supports the OOP principle "encapsulation"
- Class members can be:
  - `public` – accessible from any class
  - `protected` – accessible from the class itself and all its descendent classes
  - `private` – accessible from the class itself only
  - `internal` – accessible from the current assembly (used by default)
  - **Protected Internal : Protected** (anywhere in derived class inside or outside the assembly)**+ Internal** (with instance of class only inside the assembly)

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)?

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



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 = null;
        age = 0;
    }
    // Constructor with parameters
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    // More code comes here …
}
```

As rule constructors should initialize all own class fields.

Constructors and Initialization

- Pay attention when using inline initialization!

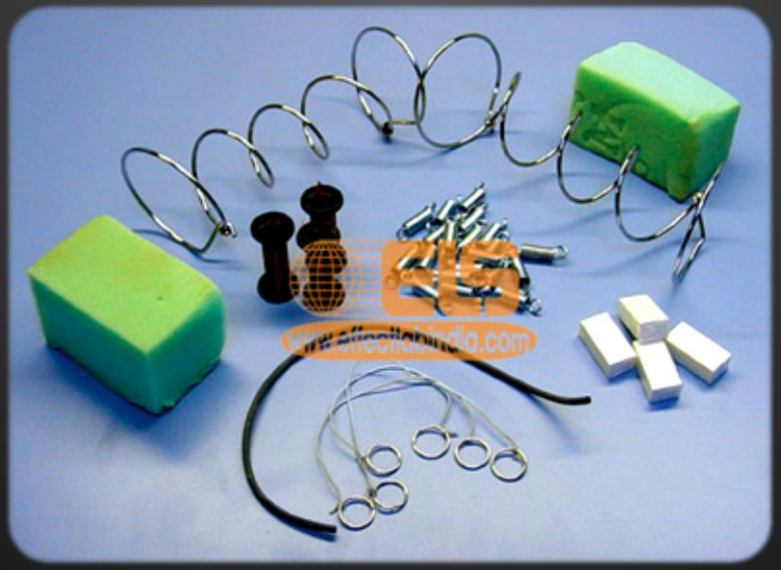
```
public class ClockAlarm
{
   private int hours = 9; // Inline initialization
   private int minutes = 0; // Inline initialization
   // Default constructor
   public ClockAlarm()
   { }
   // Constructor with parameters
   public ClockAlarm(int hours, int minutes)
   {
      this.hours = hours; 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

Chaining Constructors Calls

- Reusing the constructors

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

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

  public Point(int xCoord, int yCoord)
  {
    this.xCoord = xCoord;
    this.yCoord = yCoord;
  }
  // More code comes here …
}
```

Fields, Constants 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 85;
    public const double E = 2.71828182845904523 54;
    public const double LN10 = 2.30258509299405;
    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 ReadOnlyExample
{
  private readonly int size;
  public ReadOnlyExample(int aSize)
  {
    size = aSize; // can not be further modified!
  }
}
```

The Role of Properties

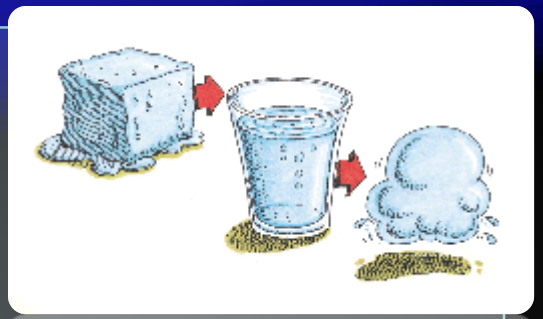
- Expose object's data to the outside world
- Control how the data is manipulated
- Properties can be:
  - Read-only
  - Write-only
  - Read and write
- Give good level of abstraction
- Make writing code easier

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, e.g. validation logic

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
};
```

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

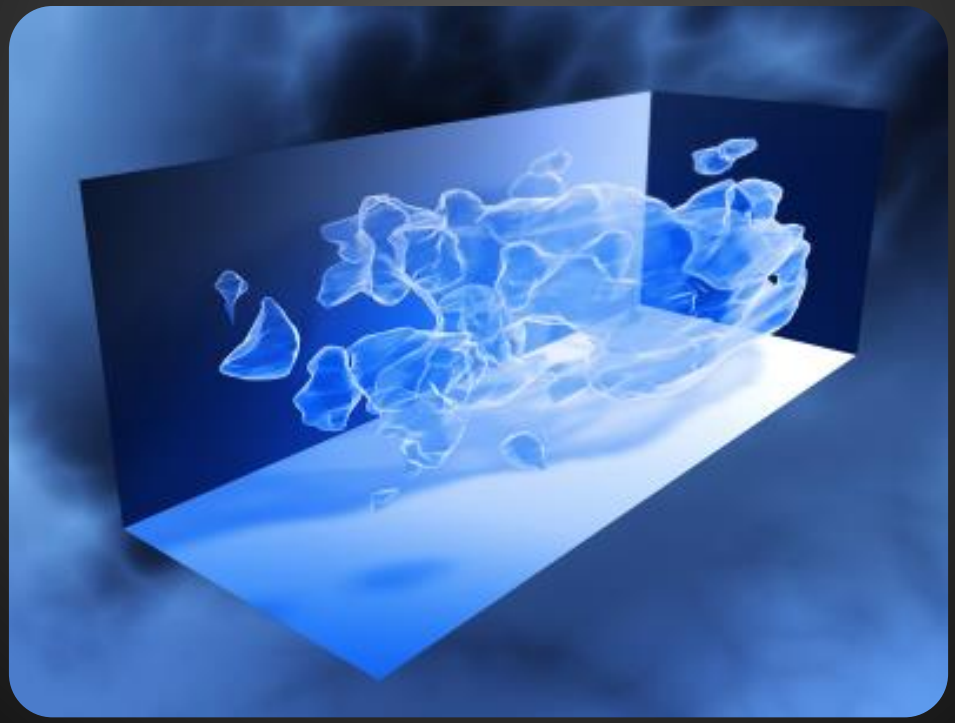
Defining Class Dog – Example

```
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



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;
}
```

What is a Namespace?

- Namespaces are used to organize the source code into more logical and manageable way
- Namespaces can contain
  - Definitions of classes, structures, interfaces and other types and other namespaces
- Namespaces can contain other namespaces
- For example:
  - `System` namespace contains `Data` namespace
  - The name of the nested namespace is `System.Data`

Full Class Names

- A full name of a class is the name of the class preceded by the name of its namespace

```
<namespace_name>.<class_name>
```

- Example:
  - `Array` class, defined in the `System` namespace
  - The full name of the class is `System.Array`

Including Namespaces

- The using directive in C#:

```
using <namespace_name>
```

- Allows using types in a namespace, without specifying their full name

  Example:

```
using System;
DateTime date;
```

  instead of

```
System.DateTime date;
```

Common Type System (CTS)

- CTS defines all data types supported in .NET Framework
  - Primitive types (e.g. `int`, `float`, `object`)
  - Classes (e.g. `String`, `Console`, `Array`)
  - Structures (e.g. `DateTime`)
  - Arrays (e.g. `int[]`, `string[,]`)
  - Etc.
- Object-oriented by design

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));
    }
}
```

Static Methods

- Static methods are common for all instances of a class (shared between all instances)
  - Returned value depends only on the passed parameters
  - No particular class instance is available
- Syntax:
  - The name of the class, followed by the name of the method, separated by dot

```
<class_name>.<method_name>(<parameters>)
```

Calling Static Methods – Examples

```
using System;

double radius = 2.9;
double area = Math.PI * Math.Pow(radius, 2);
Console.WriteLine("Area: {0}", area);
// Area: 26,4207942166902

double precise = 8.7654321;
double round3 = Math.Round(precise, 3);
double round1 = Math.Round(precise, 1);
Console.WriteLine(
    "{0}; {1}; {2}", precise, round3, round1);
// 8,7654321; 8,765; 8,8
```

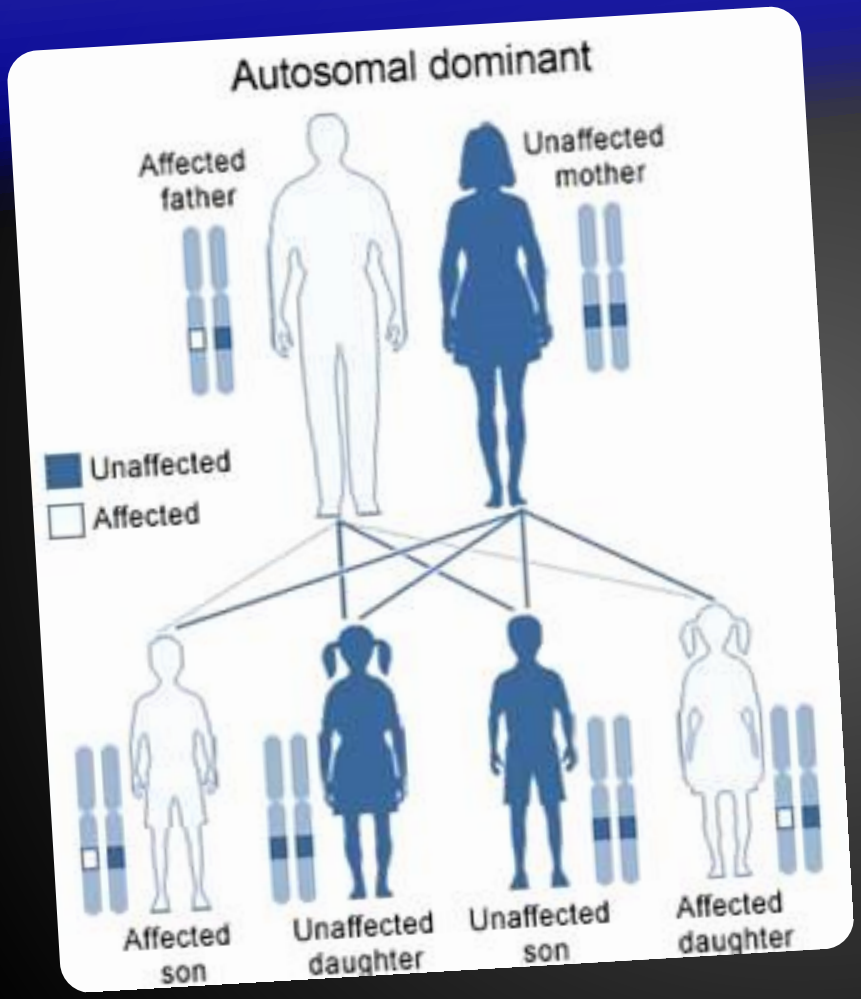
Constant field

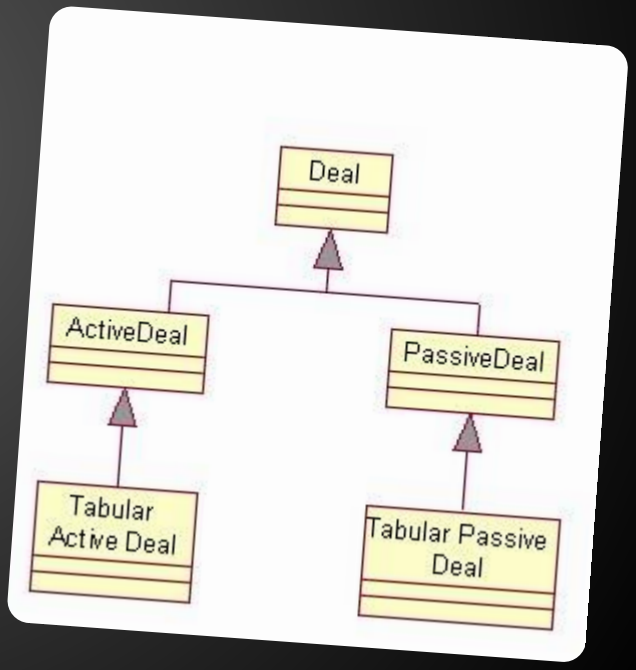
Static method

Static method

Static method

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
- 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)

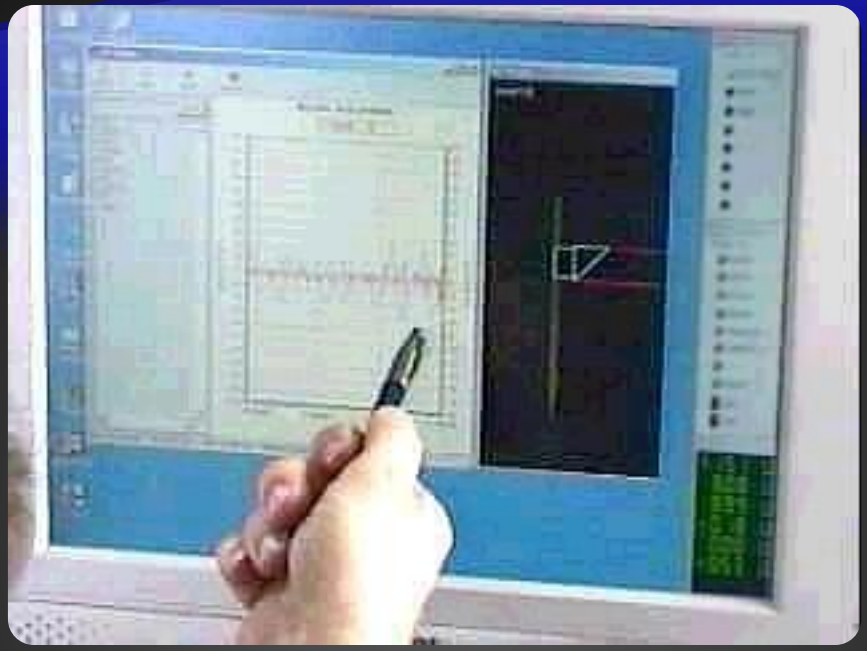
- Polymorphism is usually implemented through:
  - Virtual methods (`virtual`)
  - Abstract methods (`abstract`)
  - Methods from an interface (`interface`)
- In C# to override virtual method the keyword `override` is used

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.PrintName());
    }

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



Interfaces and Abstract Classes

Interfaces

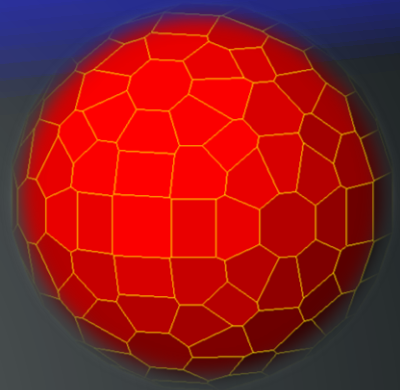
- Describe a group of methods (operations), properties and events
  - Can be implemented by given `class` or `structure`
- Define only the methods' prototypes
- No concrete implementation
- Can be used to define `abstract` data types
- Can not be instantiated
- Members do not have scope modifier and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

Interfaces – Example

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

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 can implement (support) one or many interfaces
- Interface realization must implement all its methods
- If some methods do not have implementation the `class` 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;
    }
}
```

IComparable<

- **IComparable** allows custom sorting of objects when implemented. When a class implements this interface, we must add the public method **CompareTo(T)**. We implement custom sorting for a class with IComparable.
- **CompareTo(T)** : Compares this instance with a specified object or String and returns an integer that indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified object or String.
  - **public int CompareTo (object value);**
  - **public int CompareTo (string strB);**

IComparable, CompareTo(object)

```
using System;

public class TestClass
{}

public class Example
{
   public static void Main()
   {
      var test = new TestClass();
      Object[] objectsToCompare = { test, test.ToString(), 123,
                                    123.ToString(), "some text",
                                    "Some Text" };
      string s = "some text";
      foreach (var objectToCompare in objectsToCompare) {
         try {
            int i = s.CompareTo(objectToCompare);
            Console.WriteLine("Comparing '{0}' with '{1}': {2}",
                              s, objectToCompare, i);
         }
         catch (ArgumentException) {
            Console.WriteLine("Bad argument: {0} (type {1})",
                              objectToCompare,
                              objectToCompare.GetType().Name);
         }
      }
   }
}
```

IComparable, CompareTo(string)

```
public class Example
{
   public static void Main()
   {
      string strFirst = "Goodbye";
      string strSecond = "Hello";
      string strThird = "a small string";
      string strFourth = "goodbye";

      // Compare a string to itself.
      Console.WriteLine(CompareStrings(strFirst, strFirst));

      Console.WriteLine(CompareStrings(strFirst, strSecond));
      Console.WriteLine(CompareStrings(strFirst, strThird));

      // Compare a string to another string that varies only by case.
      Console.WriteLine(CompareStrings(strFirst, strFourth));
      Console.WriteLine(CompareStrings(strFourth, strFirst));
   }

   private static string CompareStrings( string str1, string str2 )
   {
      // Compare the values, using the CompareTo method on the first string.
      int cmpVal = str1.CompareTo(str2);

       if (cmpVal == 0) // The strings are the same.
         return "The strings occur in the same position in the sort order.";
      else if (cmpVal < 0)
         return "The first string precedes the second in the sort order.";
      else
         return "The first string follows the second in the sort order.";
    }
}
```

Abstract Classes

- Abstract method is a method without implementation
  - Left empty to be implemented by some descendant class
- 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();
}
```

Abstract classes Vs Interfaces

Abstract classes can have implementations for some of its members (Methods), but the interface can't have implementation for any of its members.

Interfaces cannot have fields where as an abstract class can have fields.

An interface can inherit from another interface only and cannot inherit from an abstract class, where as an abstract class can inherit from another abstract class or another interface.

A class can inherit from multiple interfaces at the same time, where as a class cannot inherit from multiple classes at the same time.

Abstract class members can have access modifiers where as interface members cannot have access modifiers.

Abstract Class

- A class can inherit only one abstract class.
- An abstract class can provide complete, default code and/or just the details that have to be overridden.
- An abstract class can contain access modifiers for the subs, functions, properties
- An abstract class defines the core identity of a class and therefore it is used for objects of the same type
- If various implementations are of the same kind and use common behaviour or status then abstract class is better to use.
- If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.

Interface

- A class can inherit several interfaces.
- An interface cannot provide any code, just the signature.
- An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public
- Interfaces are used to define the basic abilities of a class.
- If various implementations only share method signatures then it is better to use Interfaces.
- If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.

Object-Oriented Programming with C#

Questions?