Advanced Programming C#

Lecture 7

dr inż. Małgorzata Janik

malgorzata.janik@pw.edu.pl

Winter Semester 2018/2019

C#: classes & objects

Class members

- Constructors
- Destructors
- Fields
- Methods
- Properties
- Indexers
- Delegates
- Events
- Nested Classes

Class members

- **Constructors**
- **Destructors**
- **Fields**
- **Methods**
- Properties
- Indexers
- Delegates
- Events
- Nested Classes

Example C# Class

```
// Namespace Declaration
using System;

// helper class
class OutputClass {
        string myString; // Field
        int n = 666;

        // Constructor
        public OutputClass(string inputString)
        {
                myString = inputString;
        }

        // Method
        public void printString()
        {
                Console.WriteLine("{0}", myString);
        }

        // Destructor
        ~OutputClass()
        {
           // Some resource cleanup routines
        }
}

// Program start class
class ExampleClass
{
        // Main begins program execution.
        public static void Main()
        {
                // Instance of OutputClass
                OutputClass outCl = new OutputClass("This is printed by the output class.");

                // Call Output class' method
                 outCl.printString();
        }
}
```

field

direct initialization

If not specified: **private**

constructor

method

destructor

Instance of Class

http://csharp-station.com/Tutorial/CSharp/Lesson07

Example C# Class

```
// Namespace Declaration
using System;

// helper class
class OutputClass {
        string myString; // Field
        int n = 666;

        // Constructor
        public OutputClass(string inputString)
        {
                myString = inputString;
        }

        // Method
        public void printString()
        {
                Console.WriteLine("{0}", myString);
        }

        // Destructor
        ~OutputClass()
        {
           // Some resource cleanup routines
        }
}

// Program start class
class ExampleClass
{
        // Main begins program execution.
        public static void Main()
        {
                // Instance of OutputClass
                OutputClass outCl = new OutputClass("This is printed by the output class.");

                // Call Output class' method
                 outCl.printString();
        }
}
```

pola

bezpośrednia inicjalizacja

Jeśli nie użyjemy jawnie specifikatora dostępu: **private**

konstruktor

metoda

destruktor

Instancja klasy (obiekt)

http://csharp-station.com/Tutorial/CSharp/Lesson07

Static methods

Suppose OutputClass had the following ***static*** method:

```
public static void staticPrinter()
 {
    Console.WriteLine("There is only one of me.");
 }
```

Then you could call that function from Main() like this:

```
OutputClass.staticPrinter();
```

ClassName.MethodName();

C++: ClassName::MethodName();

You must **call static class members through their class name** and not their instance name.

This means that you don't need to instantiate a class to use its static members.

There is only ever one copy of a static class member.

http://csharp-station.com/Tutorial/CSharp/Lesson07

Properties & Indexers

Class members

- Constructors
- Destructors
- Fields
- Methods
- **Properties**
- **Indexers**
- Delegates
- Events
- Nested Classes

Properties

Traditional Encapsulation Without Properties

Languages that don't have properties will use methods (functions or procedures) for encapsulation.
The idea is to manage the values inside of the object, state, avoiding corruption and misuse by calling code.

```
using System;

public class Customer
{
    private int m_id = -1;
    public int GetID()
    {
        return m_id;
    }
    public void SetID(int id)
    {
        m_id = id;
    }

    private string m_name = string.Empty;
    public string GetName()
    {
        return m_name;
    }
    public void SetName(string name)
    {
        m_name = name;
    }
}
```

„Set" and „Get" methods

An Example of Traditional Class Field Access

```
public class CustomerManagerWithAccessorMethods
{
    public static void Main()
    {
        Customer cust = new Customer();

        cust.SetID(1);
        cust.SetName("Jan Kowalski");

        Console.WriteLine(
            "ID: {0}, Name: {1}",
            cust.GetID(),
            cust.GetName());

        Console.ReadKey();
    }
}
```

http://csharp-station.com/Tutorial/CSharp/Lesson10

Properties

Encapsulating Type State with Properties

Accessing Class Fields With Properties

```
using System;
public class Customer
{
    private int m_id = -1;
    public int ID
    {
        get
        {   return m_id;
        }
        set
        {    m_id = value;
        }
    }
    private string m_name = string.Empty;
    public string Name
    {
        get
        {    return m_name;
        }
        set
        {    m_name = value;
        }
    }
}
```

See lecture 3

```
public class CustomerManagerWithProperties
{
    public static void Main()
    {
        Customer cust = new Customer();

        cust.ID = 1;
        cust.Name = "Jan Kowalski";

        Console.WriteLine(
            "ID: {0}, Name: {1}",
            cust.ID,
            cust.Name);

        Console.ReadKey();
    }
}
```

To read from/write to a property, use the property as if it were a field.

http://csharp-station.com/Tutorial/CSharp/Lesson10

Properties: read-only

Creating Read-Only Properties

```
using System;

public class Customer
{
    private int m_id = -1;
    private string m_name = string.Empty;
    public Customer(int id, string name)
    {
        m_id = id;
        m_name = name;
    }
    public int ID
    {
        get
        {      return m_id;
        }
    }
    public string Name
    {
        get
        {      return m_name;
        }
    }
}
```

Only **get**

```
public class ReadOnlyCustomerManager
{
    public static void Main()
    {
        Customer cust = new Customer(1, "Jan Kowalski");

        Console.WriteLine(
            "ID: {0}, Name: {1}",
            cust.ID,
            cust.Name);

        Console.ReadKey();
    }
}
```

http://csharp-station.com/Tutorial/CSharp/Lesson10

Properties: write-only

Creating Write-Only Properties

```
using System;

public class Customer
{
  private int m_id = -1;

  public int ID
  {
    set
    {
      m_id = value;
    }
  }

  private string m_name = string.Empty;

  public string Name
  {
    set
    {
      m_name = value;
    }
  }

  public void DisplayCustomerData()
  {
    Console.WriteLine("ID: {0}, Name: {1}", m_id, m_name);
  }
}
```

Only **set**

```
public class WriteOnlyCustomerManager
{
  public static void Main()
  {
    Customer cust = new Customer();

    cust.ID = 1;
    cust.Name = "Jan Kowalski";

    cust.DisplayCustomerData();

    Console.ReadKey();
  }
}
```

http://csharp-station.com/Tutorial/CSharp/Lesson10

Properties

Auto-Implemented Properties

The pattern where a property encapsulates a property with get and set accessors, without any other logic is common.

That's why C# 3.0 introduced a **new syntax** for a property, called an **auto-implemented property**, which allows you to create properties without get and set accessor implementations

```
using System;

public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }
}
```

```
public class AutoImplementedCustomerManager
{
  static void Main()
  {
    Customer cust = new Customer();

    cust.ID = 1;
    cust.Name = "Jan Kowalski";

    Console.WriteLine(
      "ID: {0}, Name: {1}",
      cust.ID,
      cust.Name);

    Console.ReadKey();
  }
}
```

http://csharp-station.com/Tutorial/CSharp/Lesson10

Indexers

Indexers allow your class to be used just like an array.

An Example of An Indexer: IntIndexer

```
using System;
class IntIndexer
{
  private string[] myData;

  public IntIndexer(int size)
  {
    myData = new string[size];

    for (int i=0; i < size; i++)
    {
      myData[i] = "empty";
    }
  }

  public string this[int pos]
  {
    get
    {
      return myData[pos];
    }
    set
    {
      myData[pos] = value;
    }
  }
```

Indexer is identified by the **this** keyword and **square brackets:**

this[int pos]

Implementation of an Indexer is the same **as a Property**

```
static void Main(string[] args)
  {
    int size = 10;

    IntIndexer myInd = new IntIndexer(size);

    myInd[9] = "Some Value";
    myInd[3] = "Another Value";
    myInd[5] = "Any Value";

    Console.WriteLine("\nIndexer Output\n");

    for (int i=0; i < size; i++)
    {
      Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
    }
  }
}
```

Indexer

Effect?

Using an integer is a common means of accessing arrays in many languages, but the C# Indexer goes beyond this.
Indexers can be declared with multiple parameters and each parameter may be a different type.
Additional parameters are separated by commas, the same as a method parameter list.
Valid parameter types for Indexers include integers, enums, and strings. Additionally, Indexers can be overloaded.

http://csharp-station.com/Tutorial/CSharp/Lesson11

Indexers

Indexers allow your class to be used just like an array.

An Example of An Indexer: IntIndexer

```
using System;
class IntIndexer
{
  private string[] myData;

  public IntIndexer(int size)
  {
    myData = new string[size];

    for (int i=0; i < size; i++)
    {
      myData[i] = "empty";
    }
  }

  public string this[int pos]
  {
    get
    {
      return myData[pos];
    }
    set
    {
      myData[pos] = value;
    }
  }
```

Indexer is identified by the **this** keyword and **square brackets:**

this[int pos]

Implementation of an Indexer is the same **as a Property**

```
static void Main(string[] args)
 {
   int size = 10;

   IntIndexer myInd = new IntIndexer(size);

   myInd[9] = "Some Value";
   myInd[3] = "Another Value";
   myInd[5] = "Any Value";

   Console.WriteLine("\nIndexer Output\n");

   for (int i=0; i < size; i++)
   {
     Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
   }
 }
}
```

Indexer

```
Indexer Output
myInd[0]: empty
myInd[1]: empty
myInd[2]: empty
myInd[3]: Another Value
myInd[4]: empty
myInd[5]: Any Value
myInd[6]: empty
myInd[7]: empty
myInd[8]: empty
myInd[9]: Some Value
```

Using an integer is a common means of accessing arrays in many languages, but C# goes beyond this.
Indexers can be declared with multiple parameters and each parameter may be a different type.
Additional parameters are separated by commas, the same as a method parameter list.
Valid parameter types for Indexers include integers, enums, and strings. Additionally, indexers can be overloaded.

http://csharp-station.com/Tutorial/CSharp/Lesson11

Class Inheritance & Polymorphism

Inheritance

```
using System;

public class ParentClass
{
    public ParentClass()
    {
        Console.WriteLine("Parent Constructor.");
    }

    public void print()
    {
        Console.WriteLine("I'm a Parent Class.");
    }
}

public class ChildClass : ParentClass
{
    public ChildClass()
    {
        Console.WriteLine("Child Constructor.");
    }

    public static void Main()
    {
        ChildClass child = new ChildClass();

        child.print();
    }
}
```

What is the effect?

http://csharp-station.com/Tutorial/CSharp/Lesson08

Inheritance

```
using System;

public class ParentClass
{
    public ParentClass()
    {
        Console.WriteLine("Parent Constructor.");
    }

    public void print()
    {
        Console.WriteLine("I'm a Parent Class.");
    }
}

public class ChildClass : ParentClass
{
    public ChildClass()
    {
        Console.WriteLine("Child Constructor.");
    }

    public static void Main()
    {
        ChildClass child = new ChildClass();

        child.print();
    }
}
```

Output:

```
Parent Constructor.
Child Constructor.
I'm a Parent Class.
```

Base classes are automatically instantiated before derived classes.

http://csharp-station.com/Tutorial/CSharp/Lesson08

Inheritance

```
using System;

public class ParentClass
{
    public ParentClass()
    {
        Console.WriteLine("Parent Constructor.");
    }

    public void print()
    {
        Console.WriteLine("I'm a Parent Class.");
    }
}

public class ChildClass : ParentClass
{
    public ChildClass()
    {
        Console.WriteLine("Child Constructor.");
    }

    public static void Main()
    {
        ChildClass child = new ChildClass();

        child.print();
    }
}
```

Output:

```
Parent Constructor.
Child Constructor.
I'm a Parent Class.
```

Base classes are automatically instantiated before derived classes.

C# supports **single class inheritance only**. Therefore, you can specify only one base class to inherit from. However, it does allow multiple interface inheritance.

Inheritance: *base* keyword

```
using System;

public class Parent
{
   string parentString;

   public Parent()
   {
      Console.WriteLine("Parent Constructor.");
   }
   public Parent(string myString)
   {
      parentString = myString;
      Console.WriteLine(parentString);
   }
   public void print()
   {
      Console.WriteLine("I'm a Parent Class.");
   }
}
```

```
public class Child : Parent
{
   public Child() : base("From Derived")
   {
      Console.WriteLine("Child Constructor.");
   }
   public new void print()
   {
      base.print();
      Console.WriteLine("I'm a Child Class.");
   }
   public static void Main()
   {
      Child child = new Child();
      child.print();
      ((Parent)child).print();
   }
}
```

The colon, ":", and keyword base call the base class constructor with the matching parameter list.

Using **the base keyword**, you can access any of a base class public or protected class members.

http://csharp-station.com/Tutorial/CSharp/Lesson08

Polymorphism

- **A Base Class With a Virtual Method**

```
using System;

public class DrawingObject
{
   public virtual void Draw()
{
    Console.WriteLine("I'm just a generic drawing object.");
}
}
```

The ***virtual*** modifier indicates to derived classes that they can *override* this method.

The ***override*** modifier allows a method to override the *virtual* method of its base class at run-time.

- **Derived Classes With Override Methods**

```
using System;

public class Line : DrawingObject
{
  public override void Draw()
  {
    Console.WriteLine("I'm a Line.");
  }
}

public class Circle : DrawingObject
{
  public override void Draw()
  {
    Console.WriteLine("I'm a Circle.");
  }
}

public class Square : DrawingObject
{
  public override void Draw()
  {
    Console.WriteLine("I'm a Square.");
  }
}
```

Polymorphism

- **Program implementing polymorphism**

```
using System;

public class DrawDemo
{
   public static int Main( )
   {
      DrawingObject[] dObj = new DrawingObject[4];

      dObj[0] = new Line();
      dObj[1] = new Circle();
      dObj[2] = new Square();
      dObj[3] = new DrawingObject();

      foreach (DrawingObject drawObj in dObj)
      {
         drawObj.Draw();
      }

      return 0;
   }
}
```

Base class array

Child classes elements

Effect?

http://csharp-station.com/Tutorial/CSharp/Lesson09

Polymorphism

- **Program implementing polymorphism**

```
using System;

public class DrawDemo
{
   public static int Main( )
   {
      DrawingObject[] dObj = new DrawingObject[4];

      dObj[0] = new Line();
      dObj[1] = new Circle();
      dObj[2] = new Square();
      dObj[3] = new DrawingObject();

      foreach (DrawingObject drawObj in dObj)
      {
         drawObj.Draw();
      }

      return 0;
   }
}
```

Base class array

Child classes elements

Output:

```
I'm a Line.
I'm a Circle.
I'm a Square.
I'm just a generic drawing object.
```

http://csharp-station.com/Tutorial/CSharp/Lesson09


WYDZIAŁ FIZYKI
POLITECHNIKA WARSZAWSKA
POLITECHNIKA WARSZAWSKA
WARSAW UNIVERSITY OF TECHNOLOGY

Interfaces

Interfaces

An interface looks like a class, but has no implementation.
The only thing it contains are declarations of events, indexers, methods and/or properties.
Classes and structs that implements an interface must provide an implementation for each interface member.
The interface forces each component to expose specific public members that will be used in a certain way.

Defining an Interface

```
interface IMyInterface
{
  void MethodToImplement();
}
```

Using an Interface

```
class InterfaceImplementer : IMyInterface
{
  static void Main()
  {
    InterfaceImplementer iImp =
                    new InterfaceImplementer();
    iImp.MethodToImplement();
  }

  public void MethodToImplement()
  {
   Console.WriteLine("MethodToImplement() called.");
  }
}
```

http://csharp-station.com/Tutorial/CSharp/Lesson13

Interfaces

An interface looks like a class, but has no implementation.
The only thing it contains are declarations of events, indexers, methods and/or properties.
Classes and structs that implements an interface must provide an implementation for each interface member.
The interface forces each component to expose specific public members that will be used in a certain way.

Defining an Interface

```
interface IMyInterface
{
  void MethodToImplement();
}


interface IMyInterface2
{
  int MethodToImplement2_1(int a);
  void MethodToImplement2_1();
}
```

Using multiple Interfaces

```
class InterfaceImplementer : IMyInterface, IMyInterface2
{
  static void Main()
  {
    InterfaceImplementer iImp =
                          new InterfaceImplementer();
    iImp.MethodToImplement();
  }

  public void MethodToImplement()
  {
    Console.WriteLine("MethodToImplement() called.");
  }
  public int MethodToImplement2_1(int a)
  {
    return a;
  }
  public void MethodToImplement2_2()
  {
    Console.WriteLine("MethodToImplement2() called.");
  }

}
```

http://csharp-station.com/Tutorial/CSharp/Lesson13

Polymorphism

- **An Interface**

```
using System;

public interface IDrawingObject
{
   public void Draw();
}
```

We write only function declarations in the interface definition.

- **Classes implementing an interface**

```
using System;

public class Line : IDrawingObject
{
  public void Draw()
  {
    Console.WriteLine("I'm a Line.");
  }
}

public class Circle : IDrawingObject
{
  public void Draw()
  {
    Console.WriteLine("I'm a Circle.");
  }
}

public class Square : IDrawingObject
{
  public void Draw()
  {
    Console.WriteLine("I'm a Square.");
  }
}
```

Polymorphism

- **Program implementing polymorphism**

```
using System;

public class DrawDemo
{
   public static int Main( )
   {
      IDrawingObject[] dObj = new IDrawingObject[3];

      dObj[0] = new Line();
      dObj[1] = new Circle();
      dObj[2] = new Square();


      foreach (IDrawingObject drawObj in dObj)
      {
         drawObj.Draw();
      }

      return 0;
   }
}
```

Array of classes implementing the same interface

Classes implementing interface

Effect?

http://csharp-station.com/Tutorial/CSharp/Lesson09

Polymorphism

- **Program implementing polymorphism**

```
using System;

public class DrawDemo
{
   public static int Main( )
   {
      IDrawingObject[] dObj = new IDrawingObject[3];

      dObj[0] = new Line();
      dObj[1] = new Circle();
      dObj[2] = new Square();


      foreach (IDrawingObject drawObj in dObj)
      {
         drawObj.Draw();
      }

      return 0;
   }
}
```

Array of classes implementing the same interface

Classes implementing interface

Output:

```
I'm a Line.
I'm a Circle.
I'm a Square.
```

http://csharp-station.com/Tutorial/CSharp/Lesson09


WYDZIAŁ FIZYKI
POLITECHNIKA WARSZAWSKA
POLITECHNIKA WARSZAWSKA
WARSAW UNIVERSITY OF TECHNOLOGY

enum

enum

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum Months : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
```

```
Days today = Days.Monday;
int dayNumber =(int)today;
Console.WriteLine("{0} is day number #{1}.", today, dayNumber);

Months thisMonth = Months.Dec;
byte monthNumber = (byte)thisMonth;
Console.WriteLine("{0} is month number #{1}.", thisMonth, monthNumber);

// Output:
// Monday is day number #1.
// Dec is month number #11.
```

Task

Task can be found in a separate file on the webpage.

References

C# Station http://csharp-station.com/

Examples shown today taken from:
http://csharp-station.com/Tutorial/CSharp/Lesson07
http://csharp-station.com/Tutorial/CSharp/Lesson08
http://csharp-station.com/Tutorial/CSharp/Lesson09
http://csharp-station.com/Tutorial/CSharp/Lesson10
http://csharp-station.com/Tutorial/CSharp/Lesson11
http://csharp-station.com/Tutorial/CSharp/Lesson13
http://csharp-station.com/Tutorial/CSharp/Lesson17

THE END

dr inż. Małgorzata Janik

malgorzata.janik@if.pw.edu.pl