# Lecture 4. Interfaces

## Programming II

School of Business Informatics
Autumn 2016

*(: Walking on water and developing software from a specification are easy if both are frozen :)*

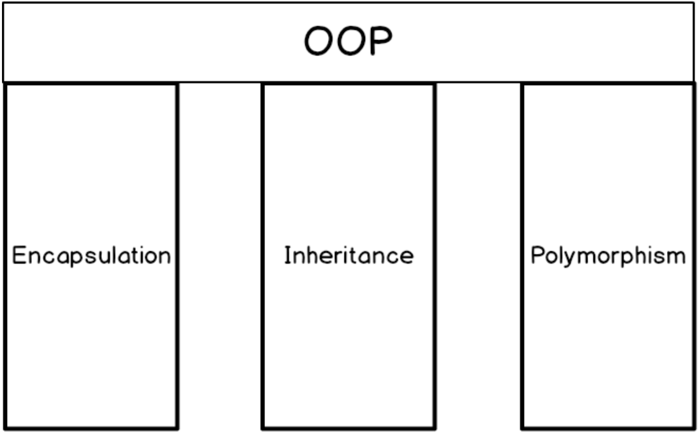
# Why OOP?

What do developers usually want from their own software?

- Easy testing and maintainability
- Adaptiveness to changes in requirements
- Maximal reuse of existing code

Object-oriented programming greatly satisfies all of these demands. It allows to develop programs of **loosely coupled components.**

# Three pillars of OOP



# Encapsulation

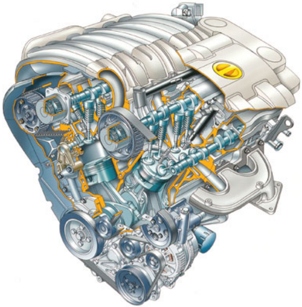
- Encapsulation requires clear separation between a class interface and its implementation
- The interface is made public while implementation details are hidden inside the class
- As a result, encapsulation **hides complexity** of the class, **ensures its integrity** and **simplifies making changes** in code

# Encapsulation example 1

Car from a driver's perspective:

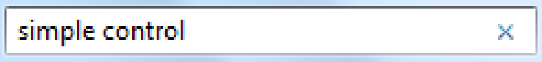
Interface

Implementation



# Encapsulation example 2

Textbox:



simple control

Internally implemented in the Textbox class:

- Blinking cursor
- Displaying text
- Scrolling
- Selecting text
- and many other features

# Encapsulation

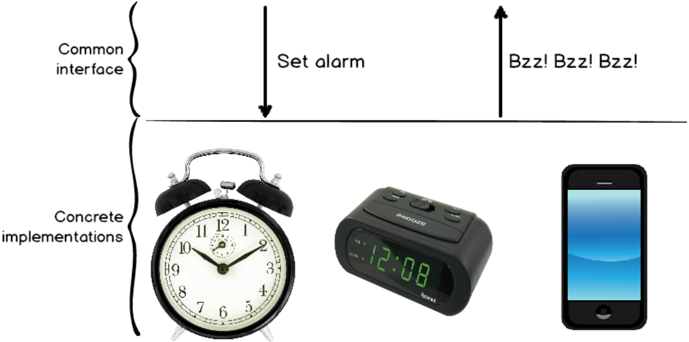
### Key principle of encapsulation

A class should provide such an interface that would make it easy for other classes to use it.

An interface is formed by public methods, properties, constructors and events.

# Polymorphism

One interface - many implementations.



Common interface

Set alarm

Bzz! Bzz! Bzz!

Concrete implementations

# Abstract programming

## Key principle

When logic is likely to change over time, program to an abstraction rather than a concrete implementation

- Last lecture: use delegates instead of direct method references
- **Today**: use interfaces rather than concrete classes
- Next time: apply inheritance hierarchies and abstract classes instead of concrete classes

# Interfaces

- An interface is formed by a group of related functions
- An interface is a **contract**. When a class references an interface it "signs an agreement" to implement all members of the interface

CONTRACT
THIS AGREEMENT made this day of 20
by
and between
and
WITNESSETH: That in consideration
kept and performed on the part
(First Party)
(Second Party),
covenants and agreements to be
hereto, respectively as herein stated:
1. Said party of
ants and agrees that it shall:
d said party of the second

# Interfaces in C#

- An interface in C# can include:
    - Properties
    - Methods
    - Events
    - Indexers
- All members of the interface are public and need to be made public in classes

# Example

```
interface IRegularPolygon
{
    int NumberOfSides { get; }
    double SideLength { get; set; }

    double Perimeter();
    double Area();
}
```

Notice the convention: interface name begins with I

# Class implementing an interface

```
class Triangle : IRegularPolygon
{
    // All members of IRegularPolygon MUST
    // be implemented here
}

class Square : IRegularPolygon
{
    // All members of IRegularPolygon MUST
    // be implemented here
}
```

# Polymorphism in action

It is possible to declare variables of an interface type and assign them to objects of concrete classes implementing the interface

```
IRegularPolygon p = new Triangle(10);
// Calling methods on an interface
p.Perimeter();    // Calls the triangle perimeter method

p = new Square(10);
p.Perimeter();    // Calls the square perimeter method
```

# Key thing to remember

- Behind an interface variable there is **always** an object of a concrete class
- **Interfaces can not be instantiated**

# Interface hierarchy

```
// Interface for a polygon on screen
interface IPolygonOnScreen : IRegularPolygon
{
    int CenterX { get; set; }
    int CenterY { get; set; }

    Color ForeColor { get; set; }
    void Draw();
}

class PolygonOnScreen : IPolygonOnScreen
{
    // Members of both IPolygonOnScreen and
        IRegularPolygon must be implemented here
}
```

# Pitfall

Don't overuse interfaces:

```
// No point in declaring this interface
interface IPerson
{
    string Name { get; set; }
    DateTime BirthDate { get; set; }
}

public class Person : IPerson
{
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}
```

# Interfaces in .NET

```
public class List<T> : IList<T>, ICollection<T>,
    IList, ICollection, IReadOnlyList<T>,
    IReadOnlyCollection<T>, IEnumerable<T>,
    IEnumerable
```

A standard List<T> implements 8(!) different interfaces. Notice the generic type specifier.

# IEnumerable and IEnumerator

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

These two interfaces are used inside the foreach loop

# IComparable

IComparable is used to provide rules for comparing custom objects:

```
public class Person : IComparable<Person>
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
    public DateTime BirthDate { get; set; }

    public int CompareTo(Person other)
    {
        return Name.CompareTo(other.Name);
    }
}
```

# Interface segregation principle

## General formula

Client code should not depend on methods that it does not use

In practice this means:

- Avoid thick interfaces containing different groups of methods in one declaration. **Build interface hierarchies.**
- Client code should use the lowest interface in the hierarchy that allows to solve the task.

# Interface hierarchies in .NET

IEnumerable:

- Iterate over a container

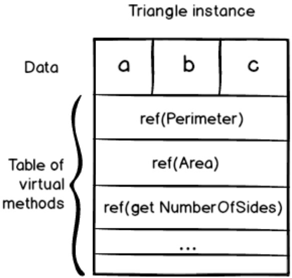
ICollection:

- Iterate over a container
- Add, remove items
- Get number of items

IList:

- Iterate over a container
- Add, remove items
- Get number of items
- Get item by index

# Physical structure of an object

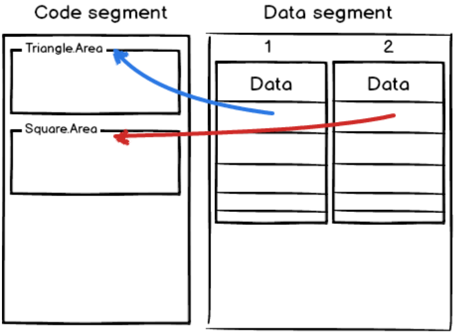


```
interface IRegularPolygon
{
   int NumberOfSides { get; }
   double SideLength { get; set; }

   double Perimeter();
   double Area();
}

class Triangle : IRegularPolygon
{
   int a,b,c;
   // Implementation of IRegularPolygon
}
```

# Memory structure



```
var polygonList = new List<IRegularPolygon>();

polygonList.Add(new Triangle(5));
polygonList.Add(new Square(5));

foreach(var p in polygonList)
    Console.WriteLine(p.Area());
```

# Design patterns

Lecture 4

Encapsulation

Interfaces in C#

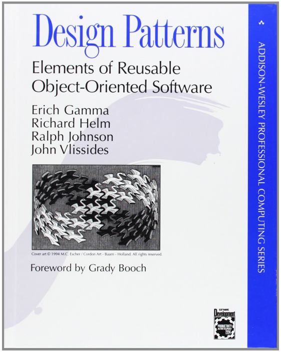
Interface hierarchies

Standard .NET interfaces

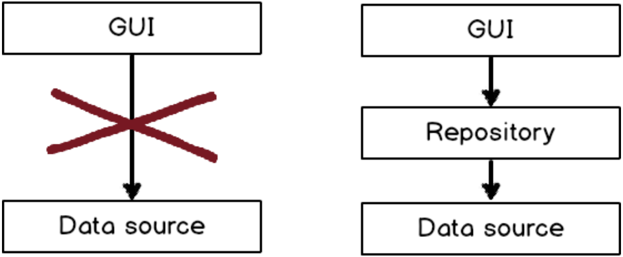
Interfaces under the hood

Repository pattern

- Over the years software developers have worked out a number of common practices, also known as patterns.
- Today there are over 30 design patterns used in different scenarios
- Using patterns simplifies software development, especially when working in a team


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# Repository pattern



GUI - graphical user interface
Data Source - file, database, remote service

## Idea of a repository

Present the data source as if it was an in-memory collection

# CRUD

A repository usually contains implementation of 4 main operations on data (CRUD):

- Create
- Read
- Update
- Delete