Lecture 5. Inheritance

Programming II

School of Business Informatics
Autumn 2016

(: What is the object-oriented way of becoming rich - Inheritance :)

Key principle

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

- use delegates instead of direct function calls
- use interfaces rather than concrete classes
- **this lecture**: apply inheritance hierarchies and abstract classes instead of concrete classes

What is inconvenient about interfaces? 3


```
1  interface IRegularPolygon
2  {
3      int NumberOfSides { get; }
4      double SideLength { get; set; }
5
6      double Perimeter();
7      double Area();
8  }
```

What is inconvenient about interfaces? 3

```
1  interface IRegularPolygon
2  {
3      int NumberOfSides { get; }
4      double SideLength { get; set; }
5
6      double Perimeter();
7      double Area();
8  }
```

Same method implementations are repeated in classes

"Is a" relationship

Inheritance **can be** applied when two entities have an "is a" relationship. Examples:

- A square **is a** shape
- A button **is a** UI control
- A keyboard **is an** input device

"Is a" relationship

Inheritance **can be** applied when two entities have an "is a" relationship. Examples:

- A square **is a** shape
- A button **is a** UI control
- A keyboard **is an** input device

Compare to "has a" relationship for composition.

Key idea

Inheritance can show its real power only in polymorphic scenarios when different implementations (contained in child classes as overriden members) are accessed through a common interface (defined in base class as virtual/abstract members)

Applying inheritance

- The primary benefit of inheritance is **code reuse**. It allows to define new classes based on existing ones without reimplementing all logic from scratch.
- In the child class all public and protected members of the base class are available
- A child class can also redefine its base class methods/properties and add new ones

Terminology


```
1  class B : A
2  {
3  }
```

- Class A - base class, parent class, superclass
- Class B - derived class, child class, subclass
- Class B derives from A

Inheritance syntax 1. Defining a child class


```
1   class RegularPolygon
2   {
3       public RegularPolygon(int numberOfSides, double
            sideLength)
4       {
5           // Do assignments
6       }
7   }
8   class Triangle : RegularPolygon
9   {
10      public Triangle(double sideLength) : base(3,
            sideLength)
11      {
12
13      }
14  }
```

Inheritance syntax 2. Overriding base class members


```
1   class RegularPolygon
2   {
3       public virtual double Area()
4       {
5           throw new NotImplementedException();
6       }
7   }
8
9   class Triangle : RegularPolygon
10  {
11      public override double Area()
12      {
13          return Math.Sqrt(3) / 4 * SideLength *
                SideLength;
14      }
15  }
```

C# inheritance details

- **protected** members of the base class are visible from all child classes, but are inaccessible from external classes
- Properties can also be made virtual and then overriden in child classes
- An event declared inside the base class cannot be called from child classes

Inheritance hierarchy in .NET

All classes in .NET derive from a common base class - System.Object (object), which has the following members:

- ToString - converts an object to a string representation. When not overridden, returns type name

Inheritance hierarchy in .NET

All classes in .NET derive from a common base class - System.Object (object), which has the following members:

- ToString - converts an object to a string representation. When not overridden, returns type name
- GetType - returns type of an object

Inheritance hierarchy in .NET

All classes in .NET derive from a common base class - System.Object (object), which has the following members:

- ToString - converts an object to a string representation. When not overridden, returns type name
- GetType - returns type of an object
- Equals - compares an object with another object. By default references are compared for reference types and values for value types

Inheritance hierarchy in .NET

All classes in .NET derive from a common base class - System.Object (object), which has the following members:

- ToString - converts an object to a string representation. When not overridden, returns type name
- GetType - returns type of an object
- Equals - compares an object with another object. By default references are compared for reference types and values for value types
- GetHashCode - calculates hash for an object

Inheritance hierarchy in .NET

All classes in .NET derive from a common base class - System.Object (object), which has the following members:

- ToString - converts an object to a string representation. When not overridden, returns type name
- GetType - returns type of an object
- Equals - compares an object with another object. By default references are compared for reference types and values for value types
- GetHashCode - calculates hash for an object

It also has a static method ReferenceEquals which compares two references for equality

Equals and the "==" operator

- The "Equals" method can be **overriden**. The implementation is resolved at runtime
- The "==" operator can be **overloaded**. The implementation is resolved at compile time.

MSDN Article on best practices of using Equals and the "==" operator

Abstract and sealed classes

- An **abstract** class can not be instantiated. Instead child classes must be defined
- **Sealed** classes can not be inherited
- **abstract** and **sealed** can be applied to individual methods

Multiple inheritance

A class in C# can have only one parent class but can implement any number of interfaces

```
1  class Child : Parent, Interface1, Interface2, ...,
       InterfaceN
2  {
3  }
```

Interface vs abstract class

Interface	**Abstract class**
Multiple inheritance	Only one base class
The interface itself does not contain any implementation. All members of the interface have to be implemented in each class	Common members can be implemented once in the base class

Best practices

- If several child classes have the same implementation of a method, consider moving the method to the base class
- If a method/property references only fields of a base class it should be declared in a base class
- Fields of the base class should be initialized in the base class constructor

- There are few special cases, when inheritance perfectly fits as the mechanism, e.g. GUI libraries or stream IO
- Most real problems are hard to be formalized using inheritance principles, thus composition is preferred over inheritance
- Inheritance breaks encapsulation - in many cases a derived class needs to know the internal organization of its base class