Programing Using C#, II

Inheritance, and Introduction to Polymorphism

Agenda:

- Inheritance, base class, subclass
- “is a” and “has-a” relations between objects.
- Introduction to polymorphism
- Static and dynamic binding
- Method overriding

Farid Naisan, farid.naisan@mau.se, University Lecturer, Malmö University

Introduction

- Inheritance is one of the most important principles of the OOP.
- Inheritance allows software reusability by creating new classes from existing ones
  - Absorb existing class’s data and behaviors.
  - Enhance with new capabilities.
- A class, a *subclass*, gets access through inheritance to the public and protected members of the class, *base class* it inherits.

Farid Naisan, farid.naisan@mau.se, University Lecturer, Malmö University

Subclass and base class

- A subclass extends a base class and make more specialized group of objects.
- A base class is also called a super class.
- Every object in C#, including those that we programmers write, inherit implicitly from the super object called Object.
- A subclass is said to be *derived* from a base or super class.

Inheritance- Syntax

- In C# the colon character ':' is used to denote inheritance.
- In this example, the class Employee inherits the class Person..

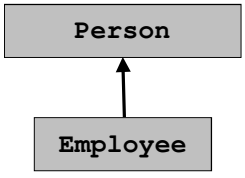
```
//"Is a" relation - inheritance
public class Employee : Person
{
    private double salary;

    //Code specialized for an employee

}
```

Base class and sub class

- Inheritance can be best explained by examples. Assume that we will write a program that stores and handles data about persons.
- A person can be an employee, a Chief or a Customer.
- Begin with a high-level abstraction and determine the very common data and operations for the group of Person : All person types have a first name, a last name and an address.
- We write a class **Person** that handles the common data..
- Every Person type, **Employee**, **Chief**, **Customer**, can use the **Person** class but they can also have their own non-common and particular fields and methods.

Employee inherits Person

- An Employee has all a Person has but also en salary (among other properties). We write a class that reuses the Person class wholly and in addition handles all about an Employee.
- The class Employee inherits the class Person, to get access to all public and protected members of the Person class without any rewriting of code.



- Person becomes the base class and Employee the subclass.
- Inheritance is indicated by an arrow with a closed and filled cap directing towards the base class.

Inheritance- example

```
public class Person
{
    private string m_firstName;
    private string m_lastName;
    private DateTime m_dateOfBirth;

    private Address m_address;  //"has a" relation - aggregation

    Properties
    // other fields,
    // constructors
    // methods

    public string FullName()
    {
        return string.Format("{0}, {1}", m_lastName.ToUpper(), m_firstName);
    }
}
```

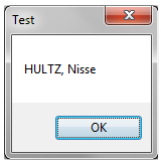
```
//"Is a" relation - inheritance
public class Employee : Person
{
    private double salary;

    //Code specialized for an employee

}
```

- **Employee** inherits **Person**. **Employee** is a subclass or a derived class (or child class). Person is a baseclass or a parent class..
- All object of Employee has access to Person's members through inheritance.

Testing the example

- **m_employee** is an object of the Employee class, but it has access to its base class' members.
- The method **FullName** is in the class **Person** but **m_employee** calls it in the same way as it were a part of its own class: **m_employee.FullName**

```
public class Test
{
    private Employee m_employee = new Employee();

    //other code

    public void TestTheEmployee()
    {
        m_employee.FirstName = "Nisse";
        m_employee.LastName = "Hultz";

        MessageBox.Show(m_employee.FullName(), "Test");
    }
}
```



Inheritance is described by a “is a” relationship

- The relationship between a base class and an inherited class is called an **“is a”** relationship. Simply put, inheritance and "is a" relationship say the same thing.
  - A grasshopper “is a” insect.
  - A dog “is a” mammal.
  - A car “is a” vehicle.
- The base class is a general class. The derived class is a specialized class and has:
  - all of the characteristics of the general object, plus
  - additional characteristics that make it special.
  - A dog has a tail and 4 legs – not all mammals do that.

Example

- Assume that we have determined to group objects as Person, Address, Student, Employee, Customer, Contact, Teacher, Chief, Librarian to write an application for.
- The next job is to associate these objects so they can effectively collaborate to make the application work. In other words, set dependencies.
- Inheritance or aggregation? That is the question.
- As a good help, we can test by the **"is a"** and **"has a**" concepts.

The ”is a” relation

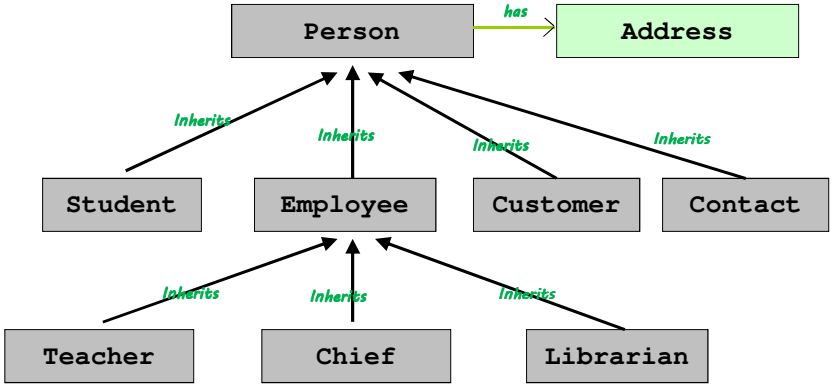
- To decide if inheritance is applicable between two types of objects, try the "is a" principle:
  - Is ” type A” a ”type B”?
- Let’s try:
  - Is an **Employee** a **Person**? Answer: Yes
    - **Employee** may inherit **Person.**
  - Is a **Contact** a **Person**? Answer: Yes
    - **Contact** may inherit **Person.**
  - Is a **Chief** en **Employee**? Answer: Yes
    - **Chief** may inherit **Employee.**
  - Is an **Address** any of the above objects Answer: No
    - Inheritance is not suitable.

“is a” relation

- A possible modeling of the last example using inheritance between objects can be done as follows:
  - **Person** becomes a *base class* to **Employee** and **Contact,** and these two in turn become subclasses to **Person**.
  - **Employee** becomes base class to **Chief** which in turn becomes a sub class to **Employee**.
  - **Chief** inherits **Employee** which inherits **Person**.
- A Chief is an Employee and a Chief is a Person.

Class diagram

- Begin with a high level of abstraction and then narrow the abstraction angle to create more specialized types within the same group of objects as in the figure below.



Try the opposite way

- Try the other way around and the answers should be no!
  - Is a **Person** an **Employee** – No (not always)! Therefore, **Person** should not inherit **Employee**.
  - Is an **Employee** a **Chief** – No (not always)! Therefore, **Employee** should not inherit Chief.
- In both of the above examples, inheritance works well in the opposite direction, as examined in the previous slide.

“Has a” relation

- When inheritance (”is a” relation) does not apply, consider Aggregation (”has a”).
  - The ”has a” relation is a type of association which describes the case when an object has another object as its part.
    - A car has 4 wheels.
- Let’s continue with the previous example.
  - ***Is*** an **Address** a **Person**? Answer: No
  - Inheritance is not suitable and should **not** be implemented here!
- ***Has*** a **Person** an **Address**? Answer: Yes
  - **Aggregation**: Person can use Address as its part, i.e. as its field.

"Is a" and "has a" example

```
public class Person
{
    private string m_firstName;
    private string m_lastName;
    private DateTime m_dateOfBirth;

    private Address m_address;  //"has a" relation - aggregation
```

Person "has an" Address

Employee "is a" Person

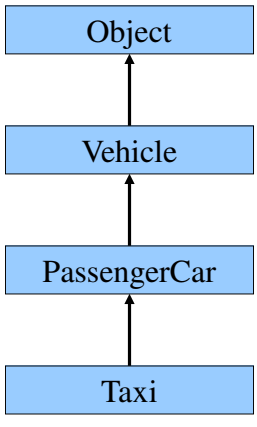
```
//"Is a" relation - inheritance
public class Employee : Person
{
    private double salary;

    //Code specialized for an employee

}
```

Types of inheritance

- The class Chief inherits Employee that inherits Person.
- Another way of saying it is:
- A Chief object "*is a*" Employee which "*is a*" Person
- There are two types of inheritance
  - Single inheritance where one object inherits only one objects
  - Multiple inheritance where one object inherits more than one objects.
- Multiple inheritance is not supported by C#.

Inheritance chain

- A base class can be derived by many sub classes.
  - The Person class can be derived by Customer, SalesMan, Student, Chief and many more classes. All of these can access Person's members.
    - customerObj.FullName
    - chiefObj.FullName
    - studentObj.FullName
- A subclass can serve as base class to other subclasses and this way build an inheritance hierarchy.



Inheritance and Constructors

- Constructors are not inherited.
- When a derived class is instantiated, the base class default constructor is executed first.
- The *base* keyword refers to an object's base class.
- The base statement that calls the base class constructor may be written only in the derived class's constructor header.
- You cannot call the base class constructor from any other method.

Calling the Base Class Constructor

- If a parameterized constructor is defined in the base class,
  - the base class must provide default constructor, or
    - the derived classes must provide a constructor, and call a base class constructor (as in the example on the next slide).
- If there is a parameterized constructor but no default constructor, the compiler will generate an error if the base class's constructor is not called in the subclass.
- Generally, when a class A is a field in another class B, it is more practical to provide a default constructor in the class A.

Calling base class' constructor

- A base class' constructor must be called if it contains constructors with parameter but no default constructor (parameterless).
- The keyword **base** is used to call the base class' constructor.

```
public class Chief : Employee
{
    //Constructor calling base class' constructor
    public Chief(double startSalary) : base(startSalary)
    {
        //Code here
    }

}
```

Note: value passing - no declaration, could pass 10000.0 instead!

Overriding Base Class Methods

- When a base class method's implementation does not work for a subclass, it can write a new implementation for the method.
- The derived class must define a method with the same signature as the base class method.
- This is known a *method overriding*. The derived class method *overrides* the base class method.
- The method must be declared as virtual in the base class and override in the derived.

"virtual" methods

- A virtual method is a method that is declared with the modifier virtual.

```
public virtual int VirtualTest()
{
    //code
    return 0;
}
```

- While base classes may (or may not) provide implementation of the method, subclasses can choose to have their own implementation for the same method.
- In this way every subclass can have its own more specialized version of the same method in the base class.

To call a base class' virtual method

- When a subclass has overridden a method, it is still possible to call the base class' virtual method.

base.MethodName()

```
public override string ToString()
{
    return "Information about " + base.ToString();
}
```

Overriding Base Class Methods

- Recall that a method's *signature* consists of the method's name, plus the list of the data types for the method's parameters in the order that they appear.
- A derived class method that overrides a base class method must have exactly the same signature as the base class method.
- It will be the derived classes' version of a virtual that will be invoked, not the base classes'.
- It a derived class does override a virtual method, then it will of course be the base class' virtual method that will be invoked.

Differentiate between overriding and overloading

- Overriding means that a derived class provides a new implementation for a method in the base class. It involves always inheritance (and at least two classes).
  - It has a syntax to follow:
    - The method must be declared as virtual in the base class
    - The method must be declared as override in the subclass
    - The method must have exactly the same signature in both the base and sub classes.
- Overloading means to define more than one method in a class with the same name. It involves only one class.
  - No syntax to follow but it has special rules: The methods must differ by:
    - either number of parameters,
    - or order of parameter types ((double, int) vs (int, double))
  - The type of the return value does not paly any role.

The ToString Method

- All C# classes are ultimately derived from a class named System.**Object**, whereby they can access its **public** and **protected** members.
- The object class has a few members; it has a virtual method named **ToString**.
- This method returns a string with information for the object, usually not very useful.
- It is a good practice to override this method in your classes and to provide more useful information about the current object.

```
public class MyClass
{
    //this class is derived from Object.
}
```


Accessibility

- Members (fields and methods) that are declared public or protected are accessible for the sub classes.
- The access modifier protected is used only in conjunction to inheritance.
- Members of the base class that are declared private are not accessible by the subclasses. Derived class can of course access these by public methods, for instance properties.
- To add extra safety to your code, use only private fields and let even the derived classes access them using properties as other objects.

Polymorphism

- Using inheritance, a derived classes get access to all the **public** and **protected** (non-private) members of a base class.
- Using polymorphism, different objects get same behavior.
- Polymorphism is a big subject and a powerful concept in object-orientation and can be implemented through:
  - dynamic binding
  - interfaces
  - abstract classes

Dynamic binding or Late binding

- Polymorphism allows a reference variable of a base class to refer to variables of subclasses.
- This way, the type of the object can be determined at run time.

```
private Person pers;
pers = new Employee();
pers = new Chief();
private object obj = new Chief();
```

- The other way around will not work:

```
private Chief pers = new Person(); //ERROR
```

- This makes sense as the `Person` object does not contain the fields defined in the `Chief` class.

Static binding or Early binding

- Static binding is when the type of the object is known already at compilation time.
  - private Person pers = new Person();
- The reference variable pers holds an object of the reference type; i.e. pers is a Person type and it holds an object of the Person type.
- While static reference variables are connected to an object type at the compilation time, dynamic binding allows creation of different types of objects (subclasses) at run time.
  - Testing of objects bound dynamically can only be done at run time.

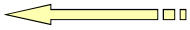
Static vs dynamic binding example

```
private void Button3_Click(object sender, EventArgs e)
{
    Person per1 = new Person(); //1
    Person per2 = new Employee();//2
    per2 = new Chief(); //3
    Salesman per3 = new Salesman(); //4
}
```

- What happens when the above code i executed?
- `per1` at (1) refers to an object of Person.
- This is called *static binding* (early binding)
  - binding at ***compile*** time.

Dynamic binding

- The same is true for line marked as 4 (see previous slide).
- `per2` refers to an object of `Employee`, to begin with.
- Then on the next line, `per2` will be referring to an object of `Chief` *at runtime*!
- This is called *dynamic binding* (late binding).
  - binding at ***runtime***.



Static vs dynamic binding

- Static binding is done at declaration. Type testing is done at compile time by the compiler.
- Dynamic binding takes places at run time and therefore no type testing can be done by the compiler. CLR will be responsible for that.
- Dynamic binding allows creation of different types of object at run time (see the example on the next slide).

Example – static and dynamic binding

pers1 and pers2 are objects of Person type at compile time, but at run time pers2 will hold an object of Employee. Fantastic!

```
private void Button3_Click(object sender, EventArgs e)
{
    Person per1 = new Person(); //1
    Person per2 = new Employee();//2
    per2 = new Chief(); //3
    Salesman per3 = new Salesman(); //4

    TestaDnyanmiskBinding(2);
    TestaDnyanmiskBinding(3);
    TestaDnyanmiskBinding(4);
}
private void TestaDnyanmiskBinding(int choice)
{
    Person pers;  //static binding to Person

    //Depending on the value of choice, pers will refer to
    //different types of object (within the hierarchy)
    switch (choice)
    {
        case 1:
            pers = new Chief();
            break;

        case 2:
            pers = new Salesman();
            break;

        case 3:
        default:
            pers = new Person();
            break;
    }
}
```


Abstract Classes

- An abstract class is one that has at least one method that is abstract
- An abstract method has only a definition and contains no implementation (no body).

```
public abstract int DoSomething(int var1, double var2);
```

- An abstract class **cannot** be instantiated, and it must be derived.
- The compiler will generate an error if you try to create an object of an abstract class (using new).
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

Interfaces

- An *interface* is similar to an abstract class in that all methods have only definitions.
  - It cannot be instantiated, and
  - all of the methods listed in an interface must be written in the subclasses.
- The purpose of an interface is to specify behavior for other classes.
- An interface looks similar to a class, except that the keyword *interface* is used instead of the keyword class, and the methods that are specified have no bodies.

```
public interface IAnimal
{
    string Name { get; set; }
    void GenerateID(int idLength, out string id);

    //other methods
}
```


More about object dependencies

- When objects depend on each other, there is a relation between them.
- These relations can be of the types:
  - Generalization
  - Association
- Generalization (inheritance)– where you begin with a class that is very general, a vehicle and build an hierarchy with less general classes.
  - ex a motor-driven vehicle inherits vehicle, a truck inherits a motor-driven vehicle.
  - Generalization is a “is a” relation.

Association

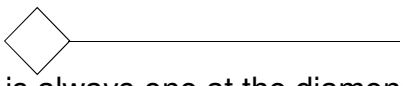
- Association – when one class is built with the help of other classes. A Contact class “has an" Address object as its field.
- “Know about” – when for example one class uses another class as method argument or return type. **Address** is not a part of MainForm but it needs to use it.

```
public class Contact : Person
{
    private Address postAddress;

    public Contact(Address adr)
    {
        this.postAddress.Copy(adr);
    }
}

public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    public void ReadAddress(out Address adr)
    {
        // code
    }
```

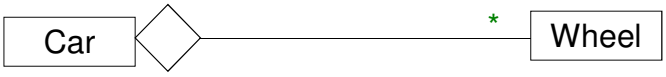

Association - aggregation

Two types of association:

- Aggregation: is a general way of denoting that something is a part of something else, a part-whole relationship.
- Aggregation is by standard shown as below:

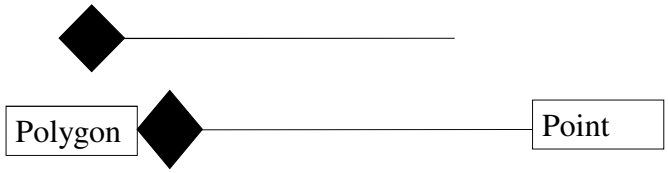


- Multiplicity is always one at the diamond side; hence, no need to write it.



Association - composition

- Composition is a strong association.
- The *whole* object strongly owns its *parts.* When the owner object (the whole) is copied or deleted the parts are also copied or deleted.
- Composition is shown as:



Association in C#

- As all objects are taken care of the Garbage Collector (GC), when an object containing another object dies, the other object is also killed by the GC if no other reference to the object exits.
- This makes it harder to decide whether it is an aggregation or composition.
- The best advice is not to worry about it and always use the term Aggregation, and use the following type of arrow to show the association (note that the arrows goes from the container to the object it contains.

An example of association in C#

```
public class Person
{
    private String firstName;
    private String surName;

    private Address homeAddress;  //Association
    private Address officeAddress;//Association

    public Person()
    {
        officeAddress = new Address();
    }
}
```


Summary

- Inheritance and polymorphism together with encapsulation form the three most important concepts in OOP. Every skillful OOP programmer has all the three principles always in mind in every solution. Do that you too!
- Inheritance and polymorphism are advanced topics. It is mostly the modeling of a problem that is difficult. C# has otherwise made it easy to implement the concepts in code.
- Both of the concepts are used very much in object-orientation.
- The main idea behind inheritance is reuse of code.
- Polymorphism aims to make different objects work similarly. It is perhaps the most powerful feature of OO.
- Polymorphism is covered in details in our next module.