Inheritance and Polymorphism with C#

Dr R. Kharel
Learning Outcome

- Understand the concept of inheritance
- Understand the concept of polymorphism
- See the use of inheritance and polymorphism with 2 case studies
Inheritance

- Building on existing classes
- **Inheritance** allows a new class to absorb an existing class’s members.
- A derived class normally adds its own fields and methods to represent a more specialized group of objects.
- Inheritance saves time by reusing proven and debugged high-quality software.
- The **direct base class** is the base class which the derived class explicitly inherits.
- An **indirect base class** is any class above the direct base class in the **class hierarchy**.
- The class hierarchy begins with class **object**.
Inheritance (contd...)

- The *is-a relationship* represents inheritance.
- For example
  - a car *is-a-kind-of* vehicle
  - and a truck *is-a-kind-of* vehicle
  - A Student *is-a-kind-of* Person
- New classes can inherit from thousands of pre-built classes in class libraries.
- Be careful with the keyword *has-a relationship*, which is not inheritance, for e.g.
  - An Aircraft has a wing
  - A Car has an engine
- This is *composition* or *aggregation* not inheritance
Base Classes and Derived Classes

- The following figure lists several simple examples of base classes and derived classes.

- Note that base classes are “more general,” and derived classes are “more specific.”

<table>
<thead>
<tr>
<th>Base class</th>
<th>Derived classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent, UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle, Triangle, Rectangle</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan, HomeImprovementLoan, MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>Faculty, Staff, HourlyWorker, CommissionWorker</td>
</tr>
<tr>
<td>BankAccount</td>
<td>CheckingAccount, SavingsAccount</td>
</tr>
</tbody>
</table>
Base Classes and Derived Classes (Cont.)

- The UML class diagram following shows an inheritance hierarchy representing a university community.
- Each arrow represents an is-a relationship.

![UML class diagram showing an inheritance hierarchy for university Community members](image)

Fig. 1. UML class diagram showing an inheritance hierarchy for university Community members.
Now consider the Shape hierarchy

We can follow the arrows to identify the *is-a* relationships.

Fig. 2. UML class diagram showing and inheritance hierarchy for Shapes
protected Members

- A base class’s **private** members are **inherited** by derived classes, but are **not directly accessible** by derived-class methods and properties.

- A base class’s **protected** members **can** be accessed by members of that base class and by members of its derived classes.
Example Point class

```csharp
using System;

public class Point
{
    private int x;
    private int y;

    public Point()
    {
    }

    public Point(int xValue, int yValue)
    {
        x = xValue;
        y = yValue;
    }

    public int X
    {
        // Set X property
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }
}
```
public int Y
{
    get
    {
        return y;
    }
    set
    {
        y = value;
    }
}

Creating the derived class

- Inheritance is all about building classes that have common information and common data that might be suitable for more than one type of class. If the Point class contained data on:
  - X position (integer)
  - Y Position (integer)
  - New attribute – Radius (double?)

To be added for a Circle (for example)
Extending the Point class

The outline of the new class might therefore be:

```csharp
public class Circle : Point
{
    //
    //

}
```

- derived-class
- base-class
using System;

public class Circle : Point
{
    private double radius;

class Circle() // Default constructor
{
}

public Circle(int xValue, int yValue, double radiusIn)
{
    // Another constructor
    X = xValue; // Use the PROPERTIES of the Point class
    Y = yValue;
    radius = radiusIn; // Radius
}

public double Radius //define getter/setter property for radius
{
    get
    {
        return radius;
    }
    set
    {
        radius = value;
    }
}
}
public class Test
{
    public static void Main()
    {
        Point p = new Point();
        p.X = 100;
        p.Y = 100;
        Console.WriteLine("Point values are " + p.X + " " + p.Y);
        Circle c = new Circle(200, 200, 20.0);
        Console.WriteLine("Circle position is " + c.X + " " + c.Y);
        Console.WriteLine("Circle diameter is " + c.Radius);
        Console.Read();// Wait for a key
    }
}
Using constructors when using inheritance

- If our **Point** base-class had a constructor

```java
public Point(int xValue, int yValue)
{
    x = xValue;
    y = yValue;
}
```
Then it would be invoked as follows:

```csharp
public class Circle : Point
{
    private double radius;
    public Circle()
    {// Default constructor
    }
    public Circle(int xValue, int yValue, double radiusIn)
    {base(xValue, yValue); // Invokes base class constructor
        radius = radiusIn; // Radius
    }
}
```

3 parameters passed to Circle constructor

2 parameters forwarded to Point base class constructor

leaving only one
Software Engineering with Inheritance

- When a new class extends an existing class, the new class inherits the members of the existing class.
- We can customize the new class to meet our needs by including additional members and by overriding base-class members.
- Independent software vendors (ISVs) can develop and sell proprietary classes.
- Users then can derive new classes from these library classes without accessing the source code.
Software Engineering with Inheritance (Cont.)

- Effective software reuse improves the software-development process
- Object-oriented programming facilitates software reuse, potentially shortening development time
- The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance
Inheritance – Case Study

- Refer separate file
Polymorphism

- **Polymorphism** enables you to write applications that process objects that share the same base class in a class hierarchy as if they were all objects of the base class.
- Polymorphism promotes extensibility.
- Means “many forms”
Polymorphism Examples

- If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` is a more specific version of a `Quadrilateral`.
- Any operation that can be performed on a `Quadrilateral` object can also be performed on a `Rectangle` object.
- These operations also can be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`.
- The polymorphism occurs when an application invokes a method through a base-class variable.
Polymorphism Examples

- As another example, suppose we design a video game that manipulates objects of many different types, including objects of classes Martian, Venusian, Plutonian, SpaceShip and LaserBeam.
- Each class inherits from the common base class SpaceObject, which contains method Draw.
- A screen-manager application maintains a collection (e.g., a SpaceObject array) of references to objects of the various classes.
- To refresh the screen, the screen manager periodically sends each object the same message — namely, Draw, while object responds in a unique way.
Area and Perimeter override base class implementation.
C# mechanism for implementing polymorphism

- Take the base class Shape

```csharp
public class Shape {
    public virtual double Area() {
        return 0.0;
    }
    public virtual double Perimeter() {
        return 0.0;
    }
}
```

Virtual - derived classes are permitted to override the method with a new implementation.

Return value is meaningless here.
Rules for virtual methods

- Allows derived classes to override
- Methods must be declared virtual to permit this
- Don’t confuse with override, which is generally used in derived classes.
- All classes derived from Object, which has a `ToString` method – don’t declare `ToString` as virtual.
public class Circle : Shape
{
    private double radius;
    private int x, y;

    public override double Area()
    {
        return radius * radius * Math.PI;
    }

    public override double Perimeter()
    {
        return 2.0 * Math.PI * radius;
    }

    public override string ToString()
    {
        return "Circle, Centre = " + x + " " + y + ", Radius = " + radius;
    }

}
Derived classes are related to base class references

- E.g.
  - Shape s = new Shape();
  - Shape s1 = new Circle();
  - Shape s2 = new Rectangle();
    - Double d = s2.Area();
  - Rectangle r = new Rectangle();
  - Circle c = new Rectangle(); (NOT PERMITTED)
Now for the power of power of polymorphism

- Create an array of Shape references
  - `Shape[] shapes = new Shape[20];`

Ask the user to instantiate the objects (Circle or Rectangle type)

```csharp
Console.WriteLine("What shape do you wish to create ?");
Console.WriteLine("Circle or Rectangle (C or R)?");
string option = Console.ReadLine();
opt = option[0];
if(opt=='C') {
    shapes[i]=new Circle();
} else {
    shapes[i]=new Rectangle();
}
i++;  // Instantiate Circle object
```
```
```
```
```
```
```
```
```
```
```
```
Build up an array of Shape objects

- Previous program loops round, building an array of shapes
- Each constructor prompts for parameters (see listing)
- Each has an Area, Perimeter and ToString method
- But which array element contains which object???
for (j = 0; j < i; j++) {
    Console.WriteLine("Shape details:" + shapes[j]);
    shapes[j].Area();
    shapes[j].Perimeter();
}

Invokes ToString method
Abstract Classes and Methods

- **Abstract classes**, or **abstract base classes** cannot be used to instantiate objects
- Abstract base classes are too general to create real objects—they specify only what is common among derived classes
- Classes that can be used to instantiate objects are called **concrete classes**
- Concrete classes provide the specifics that make it reasonable to instantiate objects
An abstract class normally contains one or more **abstract methods**, which have the keyword **abstract** in their declaration.

A class that contains abstract methods must be declared as an abstract class even if it contains concrete (nonabstract) methods.

Abstract methods do not provide implementations.
abstract property declarations have the form:

```csharp
public abstract PropertyType MyProperty
{
    get;
    set;
}
```

An abstract property may omit implementations for the `get` accessor or the `set` accessor.

Concrete derived classes must provide implementations for every accessor declared in the abstract property.
Abstract Classes and Methods (Cont.)

- Constructors and static methods cannot be declared abstract.
Abstract Classes and Methods (Cont.)

- We can use abstract base classes to declare variables that can hold references to objects of any concrete classes derived from those abstract classes.
- You can use such variables to manipulate derived-class objects polymorphically and to invoke static methods declared in those abstract base classes.
- It is common in object-oriented programming to declare an **iterator class** that can traverse all the objects in a collection.
Polymorphism – Case study

- Payroll System Using Polymorphism – Refer separate file