SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 6. Polymorphism

Outline

- Session 06. Polymorphism
  - Objectives
  - Polymorphism
  - Abstract Methods and Classes
  - Sealed Methods and Classes
  - Interfaces
  - Refactoring

SESSION 6. POLYMORPHISM

Objectives

- At the end of this session you should:
  - Understand the concept of polymorphism, how it works, and how it is used in applications;
  - Be able to apply abstract and sealed methods and classes to control the semantics of inheritance;
  - Understand how interfaces provide an alternative to inheritance and how to apply them in your programs; and
  - Know how to apply operator overloading to define extra functionality for C# operators and improve code readability.

Polymorphism

- For a programming language to be considered an object-oriented programming language, four concepts are expected:
  - Abstraction
  - Encapsulation
  - Inheritance
  - **Polymorphism**

Polymorphism

- The term polymorphism comes from the Greek words polus and morphe
  - **Polus = Many**
  - **Morphe = Forms**
- Two possible definitions (there are others)...

Polymorphism

1. ***Allows two or more objects of different types to respond to the same request***

```
class SavingsAccount : Account
{
    ...
    public override string GetStatement()
    {
            ...
    }
    ...
}

class CreditCard : Account
{
    ...
    public override string GetStatement()
    {
            ...
    }
    ...
}

class TermDeposit : Account
{
    ...
    public override string GetStatement()
    {
            ...
    }
    ...
}
```

Two or more types can respond to the same request

Polymorphism

2. ***The ability to operate on and manipulate different objects in a uniform way***

```
Account[] accounts = new Account[3];
accounts[0] = new SavingsAccount(...);
accounts[1] = new CreditCard(...);
accounts[2] = new TermDeposit(...);
...
Console.WriteLine(accounts[0].GetStatement());
Console.WriteLine(accounts[1].GetStatement());
Console.WriteLine(accounts[2].GetStatement());
```

Operate on and manipulate different objects uniformly

Polymorphism

- To use polymorphism, there are ***two key concepts:***
  - The objects to be accessed polymorphically must either:
    - Be related **via inheritance**
    - Implement a **common interface**
  - Objects are accessed using a reference **of the base class type / common interface type**

Polymorphism

- Syntax for polymorphic methods:
  - **Base class**:

```
[access_modifier ] class base_class_name
{
        [access_modifier ] virtual return_type method_name ([parameter[, ...]])
        {
                method_body
        }
        ...
}
```

  - **Derived class:**

```
[access_modifier ] class derived_class_name[ : base_class_name]
{
        [access_modifier ] override return_type method_name ([parameter[, ...]])
        {
                method_body
        }
        ...
}
```

Polymorphism

- **Binding** – how the ***computer knows/learns the memory address*** of a particular method
  - **Static binding** – the compiler determines the types of objects and thus the address
  - **Dynamic binding** – the type of object is determined at run-time
    - *Required for polymorphism*

Polymorphism

- Example: **The difference between**
  - Method hiding (static binding) and
  - Polymorphic methods (dynamic binding)

```
class BaseClass
{
    public void SimpleMethod()
    {
        Console.WriteLine("This is BaseClass.SimpleMethod()");
    }

    public virtual void PolymorphicMethod()
    {
        Console.WriteLine("This is BaseClass.PolymorphicMethod()");
    }
}
```

File: BaseClass.cs

```
class DerivedClass : BaseClass
{
    public new void SimpleMethod()
    {
        Console.WriteLine("This is DerivedClass.SimpleMethod()");
    }

    public override void PolymorphicMethod()
    {
        Console.WriteLine("This is DerivedClass.PolymorphicMethod()");
    }
}
```

Polymorphism

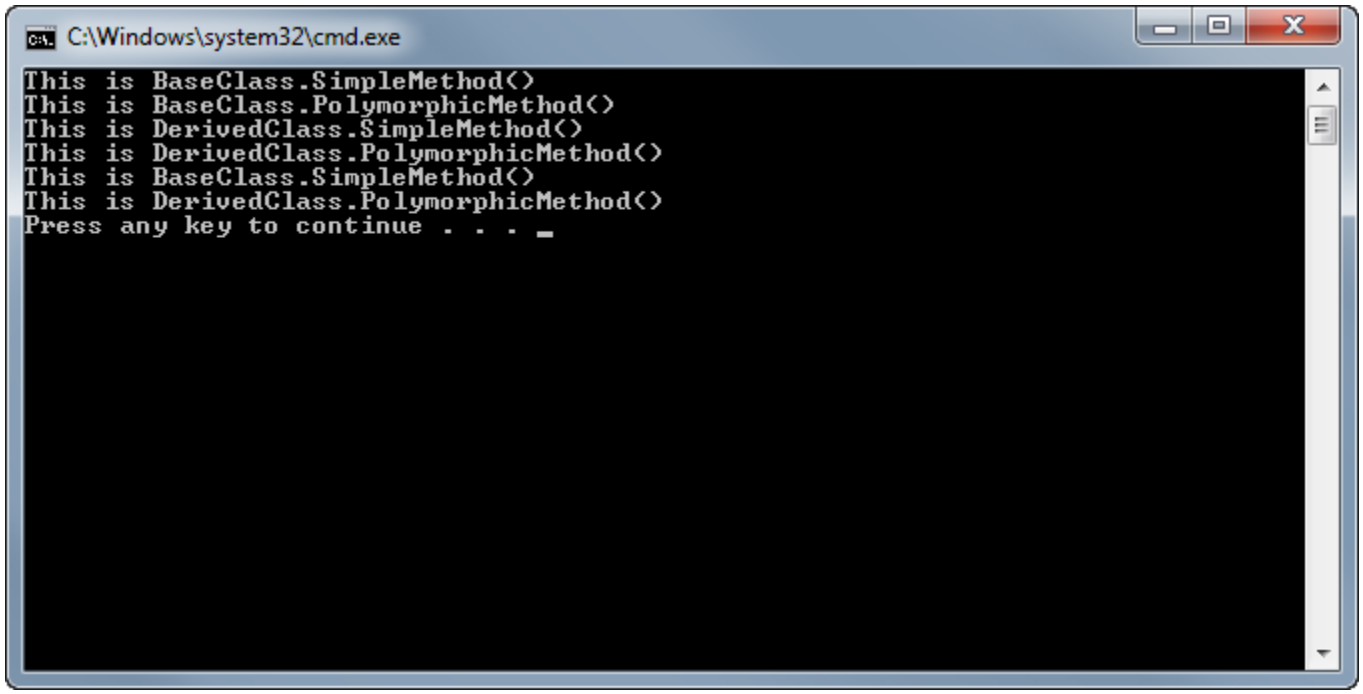
- Example: **The difference between**
  - *Method hiding* (static binding) and
  - *Polymorphic methods* (dynamic binding)

```
static void Main(string[] args)
{
    BaseClass bcObject = new BaseClass();
    DerivedClass dcObject = new DerivedClass();
    BaseClass baseReference = dcObject;

    bcObject.SimpleMethod();
    bcObject.PolymorphicMethod();

    dcObject.SimpleMethod();
    dcObject.PolymorphicMethod();

    baseReference.SimpleMethod();
    baseReference.PolymorphicMethod();
}
```

File: Program.cs

Polymorphism

- Example: **The difference between**
  - *Method hiding* (static binding) and
  - *Polymorphic methods* (dynamic binding)



Polymorphism

- It is *possible to test the data type of a class using the is operator*, e.g.,

```
foreach(Account acc in accounts)
    if(acc is CreditCard)
        Console.WriteLine("Credit card found!");
```

- It is possible to *convert between reference types using either the as operator or casting*, e.g.,

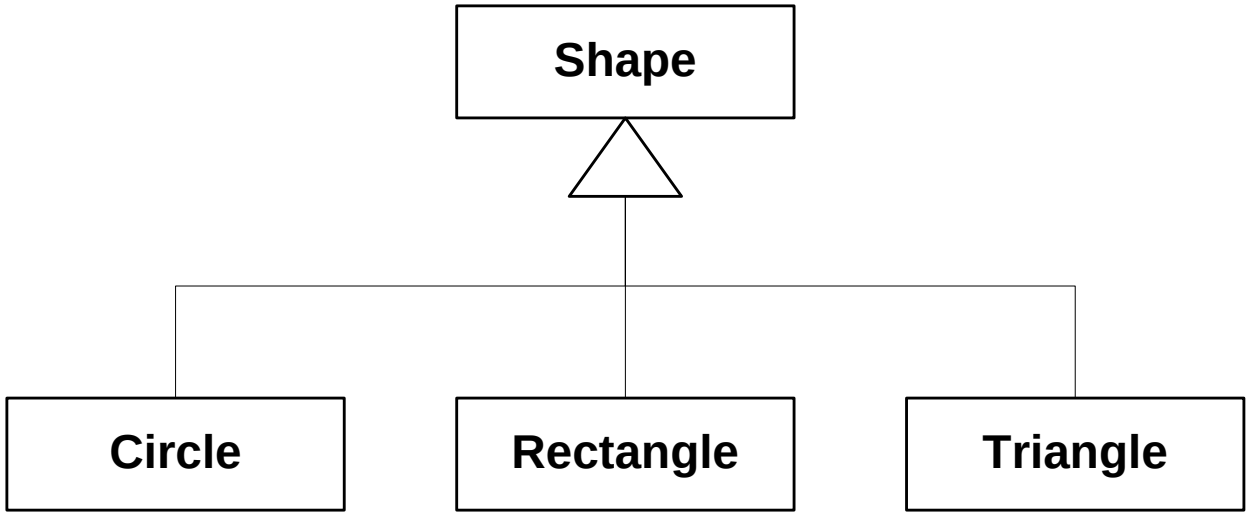
```
foreach(Account acc in accounts)
{
    if(acc is CreditCard)
    {
        CreditCard cc = acc as CreditCard;
        cc.Payment(500.00M);
    }
}
```

```
foreach(Account acc in accounts)
{
    if(acc is CreditCard)
    {
        CreditCard cc = (CreditCard)acc;

        cc.Payment(500.00M);
    }
}
```

Polymorphism

- Design guidelines:
  - **Functions defined as virtual** in a class must be applicable to all derived classes
  - Functions with no meaningful implementation at **the base class** level **must be defined as abstract**
    - Remember that this ***forces the implementation of these functions in a derived class*** before they can be instantiated
  - Base class should still ***capture attributes and functions common to all derived classes***
    - Since the *implementation of these functions does not vary between the derived classes*, they are *implemented as non-virtual*

Abstract Methods and Classes



- Consider implementing the methods
  - **GetArea()**
  - **GetPerimeter()**
- *What do you put in Shape?*

Abstract Methods and Classes

- We use abstract methods where there is ***no sensible implementation for a method***

  [access_modifier] abstract return_type method_name([parameter[, ...]]);

  - Class is also abstract (incomplete)
    - *Can be used as a base class*
    - *Cannot create instances*
    - *Can create references*
  - Derived classes abstract must implement all abstract functions *otherwise they must also be abstract*
  - Class can be abstract ***without any abstract methods***

```
[access_modifier ] abstract class derived_class_name[ : base_class_name]
{
        [access_modifier ]class_member
        ...
}
```

Sealed Methods and Classes

- Sealed method – a method *which cannot be overridden*

[access_modifier] **sealed** *return_type method_name*(*[parameter[*, ...*]]*)
{
    method_body
}

- Sealed class – a class *which cannot be inherited from*

[access_modifier] **sealed class** *base_class_name*
{
  ...
}

Interfaces

- We have already examined inheritance
  - Also known as **implementation inheritance** because the *implementation of the base class is duplicated in the derived class*

- Interfaces can similarly be described as **interface inheritance**
  - Only the *signature of public members is duplicated to the derived class*
  - Also seen as equivalent to an abstract class that only contains abstract methods

Interfaces

```
[access_modifier] interface  interface_name
{
        interface_member
        ...
}
```

- Declare properties

  ```
  type  name  { [get;] [set;] }
  ```

- Declare methods:

  ```
  return_type  method_name ([parameter[,  ...]]);
  ```

- Implementing interfaces:

  ```
  [access_modifier] class derived_class_name : interface_name
  {
      ...
  }
  ```

Interfaces

- **Rules:**
  - Names begin with '**I**', e.g., Idisposable
  - One ***interface can only define the members*** with a particular access modifier
    - If *several access modifiers are required*, one interface is required for each
  - Any *class can implement any number of interfaces in addition to inheritance*, e.g.,

```
class DerivedClass : BaseClass, ISomeInterface, IDisposable
{
   ...
}
```

Operator Overloading

- In **defining new types** it is sometimes sensible for operators to apply to those new types, e.g.,

```
invoice1 + invoice2
```

  - reads better than

```
invoice1.Add(invoice2)
```

  - or

```
Invoice.Add(invoice1, invoice2)
```

- Operator overloading allows us to define custom functionality for operators

Operator Overloading

- What operators can be overloaded?
  - Unary operators

    `+ - ! ~ ++ --`
  - Binary operators

    `+ - * / % & | ^ << >> == != > < >= <=`

- What can't be overloaded?
  - Binary operators

    `= && ||`
  - Ternary operators

    `?:`

Operator Overloading

- The syntax for overloading a unary operator is:

```
//e.g., num += 10;
public static data_type1 operatorOperatorSymbol (data_type2 rhs)
{
        ...
}

// e.g., CreditAccount = SavingAccount + 100m;
```

- Similarly, the syntax for overloading a binary operator is:

```
public static data_type1 operatorOperatorSymbol (data_type2 lhs, data_type3 rhs)
{
        ...
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Wk6Examples
{
    public abstract class Person
    {
        // attributes
        private string _FirstName, _LastName;
        // properties
        public string FirstName
        {
            get { return _FirstName;  }      set { _FirstName = value; }
        }
        public string LastName
        {
            get { return _LastName; }  set { _LastName = value; }
        }
        public abstract string GetID();
        // constructors
        public Person()  // parameter- less set to No data
        {
            _FirstName = "No First Name";
            _LastName = "Not Family Name";
        }
```

```
    public Person(string first, string last)  // custom constructor with first and last data
    {
        _FirstName = first;
        _LastName = last;
    }

    public Person(Person p)
    {
        _FirstName = p.FirstName;
        _LastName = p.LastName;
    }

    //ToString()
    public override string ToString()
    {
        return string.Format("{0}, {1}", _FirstName, _LastName.ToUpper());
    }
}
```

```
public class Customer : Person
  {
     // no attributes, properties
     private string _ID;

     // override method to implement the abstract missing from the parent class
     public override string GetID()
     {
        return _ID;
     }
     // constructor
     public Customer() : base() { _ID = ""; }
     public Customer(string id, string first, string last) : base(first, last) { _ID = id;  }
     public Customer(Customer customer) : base(customer)
     { _ID = customer.GetID(); }

     // override the Person ToString()
     public override string ToString()
     {
        return string.Format("Customer {1} - Name: {0}", base.ToString(), _ID);
     }
```

```
    // operator overloadings
    public static Customer operator+( Customer a, Customer b )
    {
        string id = ( int.Parse( a.GetID() ) + int.Parse( b.GetID() ) ).ToString();
        string first = string.Format("{0} & {1}", a.FirstName, b.FirstName);
        string last = string.Format("{0} - {1}", a.LastName, b.LastName);
        return new Customer(id, first, last);
    }
    public static bool operator==(Customer a, Customer b)
    {
        return (a.GetID() == b.GetID() && a.FirstName == b.FirstName
                                       && a.LastName == b.LastName);
    }

    public static bool operator!=(Customer a, Customer b)
    {
        return (a.GetID() != b.GetID() && a.FirstName != b.FirstName
                                       && a.LastName != b.LastName);
    }
}
```

```
public class Program
 {
    // Poly morphism: function over loading
    public static void Display(int[] array)
    {
        Console.WriteLine("\nContent of integer array:");
        foreach (int num in array) Console.Write("{0} ", num);
        Console.WriteLine();
    }

    public static void Display(decimal[] array)
    {
        Console.WriteLine("\nContent of Decimal array:");
        foreach (decimal num in array) Console.Write("{0,8:c} ", num);
        Console.WriteLine();
    }

    public static void Display(Customer[] array)
    {
        Console.WriteLine("\nContent of Customer array:");
        foreach (Customer customer in array) Console.WriteLine(customer);
        Console.WriteLine();
    }
```

```
static void Main(string[] args)
    {
      // declare testing objects
      int[] IntArray = new int[] { 6, 10, 8, 4 };
      decimal[] DecArray = new decimal[]
                                    { 12.95m, 1260m, 0.95m, 125.07m, 99.05m, 32.67m };
      Customer[] CustArray = new Customer[] { new Customer("1111", "Tom", "Anderson"),
                                              new Customer("2222", "Alice", "Wong") };
      Display(IntArray);
      Display(DecArray);
      Display(CustArray);
      Console.WriteLine("Operator Plus (+) overloading: ");
      Customer joinCustomer = CustArray[0] + CustArray[1];
      Console.WriteLine(joinCustomer);
      Console.WriteLine("Operator == overloading: ");
      if (CustArray[0] == CustArray[1])
            Console.WriteLine("{0} is the same as \n{1}", CustArray[0], CustArray[1]);
      else Console.WriteLine("{0} is NOT the same as \n{1}", CustArray[0], CustArray[1]);
```

```
        // if (joinCustomer == new Customer("3333", "Tom & Alice", "Anderson - Wong"))
        Console.WriteLine("{0} is the same as \n{1}", joinCustomer, joinCustomer);

        Customer two = new Customer("3333", "Tom & Alice", "Anderson - Wong");
        if (joinCustomer == two)
            Console.WriteLine("{0} is the same as \n{1}", joinCustomer, two);
        else
            Console.WriteLine("{0} is NOT the same as \n{1}", joinCustomer, two);
      }
   }
}
```

Refactoring

- We examined “bad smells in code” in the previous session
- There are many re-factorings to correct design problems:
  - **Extract method** – move a code fragment in one method into a separate method;
  - **Inline method** – replace calls to a very short method with the body of that method;
  - **Move method** – move a method that uses features of another class more than its own into the other class;

Refactoring

- Refactorings (cont):
  - **Remove Parameter** – remove a parameter that is no longer required by a method;
  - **Substitute algorithm** – replace a complex algorithm with a simpler algorithm;
  - **Extract class** – split a class that has become too large by moving a subset of its members into another class;
  - **Inline class** – move the members of a class that is too small into those classes that use it;
  - **Extract super-class** – move members common to two or more classes into a common base class; and
  - **Extract interface** – class members used regularly should be defined by an interface to improve reusability.

Summary

- Session 06. Polymorphism
  - Objectives
  - Polymorphism
  - Abstract Methods and Classes
  - Sealed Methods and Classes
  - Interfaces
  - Operator Overloading

Summary

- Training Videos:
  - C#: Operator Overloading