Lecture 35 − Polymorphism & Inheritance

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

So far, our discussion of object-oriented design has been limited to relatively simple classes.

Often, object-oriented design provides little benefit for "toy" applications such as the applications developed in this course.

Many of the things we have written in the class can be done with one class, or a handful of classes.

For large application programs developed by a team of software developers, object-oriented design is necessary.
  Programs will be too complex to leave unmanaged.

Coping with Evolving Software Demands

Two key characteristics are often desirable to cope with the evolving demands of software:

1 Polymorphism
   Define methods and collections that work with a variety of types
2 Extensibility
   Enhance existing types without changes to already-existing code

Not all object-oriented languages provide full support for both polymorphism and extensibility.

The word polymorphism comes from Greek and means "many forms".

Polymorphic code works for or on multiple types of object.
  The code is "generic" or flexible in what it accepts.

The same function, method, or collection can be used with many different types (without relying on type promotion).

We have already seen some examples of polymorphism in the course.

`ArrayList` is a polymorphic collection:
  We can put many different types of object in the collection.

`ArrayList.Sort` is a polymorphic function:
  It will sort the objects in the array, whatever kind of object they are.

Polymorphism exploits characteristics common to a set of types.
  Example: if two objects can be compared, they can be sorted.
  ... regardless of the actual types of the instances.

A polymorphic function operates on a related set of object types.

`ArrayList.Sort` does not need to know the type of the objects, but it does need to know how to compare the objects in the array.

A function with an `object` parameter is automatically polymorphic.
  We've seen `Equals` as an example of that.
  Hence why we need to use the `is` keyword to check the input.

A function can be polymorphic without having an `object` parameter.

Previously, we saw the `is` operator that returns true if an instance of an object is compatible with a given type.

At run-time, the `GetType( )` method returns the precise type of an object, such as `o1.GetType( );`

This may be useful when using polymorphism.

We have already done type conversion with explicit casting.
   Example: `double d = (double) x;`

The as operator allows a software developer to convert the type of an object to another compatible type at run-time.

Polymorphic functions and collections often benefit from the use of run-time type conversion.

For example, arrays of objects can be created. At run-time:
   The type of array elements can be identified.
   Entries can be converted to a compatible type for processing.

```
object[] myWishList = new object[5];
myWishList[0] = "Remote-Controlled Car";
myWishList[1] = "Porsche 911 Turbo";
myWishList[2] = 75000d;
myWishList[3] = "An easy final exam";
myWishList[4] = null;

Console.WriteLine( "\nMy WishList\n" );

for( int i = 0; i < myWishList.Length; i++ )
{
    string s = myWishList[i] as string;
    if ( s != null )
    {
        Console.Write( "{0}: ", (i + 1) );
        Console.WriteLine( s );
    }
}
```

The key statement is `string s = myWishList[i] as string;`

This will attempt a type conversion if the source and destination types are compatible.

If the type conversion is not compatible, null is assigned.
   Thus it is important to check for null after an as statement.

Inheritance: a class is based upon (or derived from) another class.

The original class is referred to as the "base" or "parent" class.
  We say the derived class "inherits" from the parent class.

What does it inherit? Fields, methods, et cetera.

Inheritance provides extensibility and reduces duplicate code.

This provides two advantages:

1 Reduce the amount of code required for objects that are related

2 Promote the consistent behaviour of related objects

In C#, a class may only inherit from one other class.
    Or, put another way, each class can have at most one parent class.

The parent class can be referred to using the base keyword.

It should be noted that some other languages (notably C++) allow inheritance from multiple base classes.

Follow the inheritance hierarchy upwards for any class in C# eventually you will reach `object`.
    If you don't explicitly declare a base class, the base class is `object`.

The syntax for inheriting from a base class is essentially the same as for implementing an interface:

attributes modifiers class *identifier* : *base*
  Example: `public class DerivedClass :  BaseClass`

Followed by the internal implementation of the derived class.

Note that the implementation of the derived class does not repeat the member function and variable declarations of the base class.
  Unless there is a need for them to have a different implementation.

```
class BaseClass
{
    // Declares a base class from which other classes
    // may be derived.  This class will automatically
    // inherit from the object class.
}

class DerivedClass : BaseClass
{
    // Declares a derived class that inherits fields
    // and functions from the BaseClass.
}
```

Let's start out with a simple class `Employee`:

```
class Employee
{
    protected string name;
    protected string title;
    protected string sin;

    public Employee( string n, string t , string s)
    {
        name = n;
        title = t;
        sin = s;
    }
}
```

Now let's add a new kind of `Employee`: the `HourlyEmployee`.

```
class HourlyEmployee : Employee
{
    double wage;
    double weeklyHours;

    public HourlyEmployee( string n, string t, string s,
                           double w, double h )
    {
        base.name = n;
        base.title = t;
        base.sin = s;
        wage = w;
        weeklyHours = h;
    }
}
```

The `HourlyEmployee` class inherits all of the fields and functions of the `Employee` class. They need not be duplicated.

It also adds two new fields only applicable to hourly employees.

The keyword base is used in the derived class to refer to the methods and fields of the base class.

Another new keyword appeared on that slide: `protected`
    Let's take a look at that on the next slide.

Access Modification

In addition to the access modifier keywords `public` and `private`, there is a third option: `protected`.

Declaring a field or method as `protected` means it is accessible only from within the class where it is declared and any derived class.

In some cases, you may not want a derived class to be able to access member fields directly; in that case, use `private`.

Using properties and the `base` keyword, it is possible to reduce the need for protected fields.

Now let's add another kind of `Employee`: the `SalaryEmployee`.

```
class SalaryEmployee : Employee
{
    double salary;

    public SalaryEmployee( string n, string t,
                           string s, double m )
    {
        base.name = n;
        base.title = t;
        base.sin = s;
        salary = m;
    }
}
```

We can use the base keyword to reduce the amount of code we have to write here by calling the constructor of the base class:

```
class SalaryEmployee : Employee
{
    double salary;

    public SalaryEmployee( string n, string t,
                           string s, double m )
                           : base (n, t, s)
    {
        salary = m;
    }
}
```

The SalaryEmployee class is closely related to HourlyEmployee.

Both classes inherit from the Employee class.
Common elements need only be defined once.

Inheritance represents the "is-a" relationship:
Hourly Employee is an Employee;
Salary Employee is an Employee.

Because of this relationship, this is legal:
```
Employee e = new HourlyEmployee( );
```

We can refer to an instance of HourlyEmployee as if it's an instance of Employee: an hourly employee is an employee.

Now suppose you wanted to add a category of employee: `Manager`.

You may decide that `Manager` is a kind of `SalaryEmployee`.

A `Manager` has a list of employees who report to him or her. Managers can also get bonuses at the end of the year.

Solution: extend `SalaryEmployee` to make `Manager`.

```
class Manager : SalaryEmployee
{
    ArrayList reports;
    double bonus;

    public Manager( string n, string t,  string s, double m,
                    double b )
    {
        base.name = n;
        base.title = t;
        base.sin = s;
        salary = m;
        bonus = b;
        reports = new ArrayList( );
    }
}
```

`Manager` inherits from `SalaryEmployee` which in turn inherits from `Employee`.

The "is-a" relationship is transitive:
    Manager is a Salary Employee;
    Salary Employee is an Employee;
    Therefore Manager is an Employee.

Because of this relationship, this is also legal:
```
Employee e = new Manager( );
```

We have already used the override keyword to implement ToString.

That is a specific example of replacing the behaviour of a base class by writing a new implementation of that method.

Suppose we add the following method to SalaryEmployee:

```
public double ComputePay( )
{
    return salary;
}
```

This method is available and can be called on instances of SalaryEmployee and Manager.

Now imagine we'd like to override this behaviour in `Manager` to account for the fact that managers can get a bonus.

```
public override void ComputePay( )
{
    return salary + bonus;
}
```

`ComputePay( )` can still be called on `Manager` and `SalaryEmployee`.

When calling `ComputePay( )` on an instance of `SalaryEmployee`, the method in `SalaryEmployee` is run.

When calling `ComputePay( )` on an instance of `Manager`, the method in `Manager` is run.
   Why? It is overidden in `Manager`.

What if the code just refers to an instance of `SalaryEmployee` and we're not sure if it's a manager or not?

At run-time, the system knows what kind of object a given instance is.

Thus, the code may refer to `Employee`, but the system knows if it is really a `HourlyEmployee`, `SalaryEmployee`, or `Manager`.

It will therefore execute the appropriate method based on the actual type of the instance.

Inheritance Plus Polymorphic Functions

Earlier we asserted that a function can be polymorphic without having an `object` parameter. Inheritance is the reason for this.

Suppose we have a method `double ComputeArea( Shape s )`.

Let's say that `Shape` is a base class and there are many derived classes: `Triangle`, `Rectangle`, `Circle`, `Polygon`...

When calling `ComputeArea`, because `Triangle` derives from `Shape` it is legal to provide a `Triangle` instance as the actual parameter.

Inheritance Plus Interfaces

A class can inherit from a base class and implement one or more interfaces concurrently.

```
class ImplClass : IFace1, IFace2
{
    // Declares a class that implements
    // two interfaces (IFace1 and IFace2).
}

class DerivedImplClass : BaseClass, IFace1, IFace2
{
    // Declares a derived class that inherits fields
    // and functions from the base class ImplClass and
    // also implements two interfaces (IFace1 and IFace2).
}
```

Because of the "is-a" relationship, if a base class implements an interface, the derived classes will benefit as well.

Example: modifying `Employee` to implement `IComparable`.
   Suppose we sort the employees by SIN (Social Insurance Number).

If implemented on `Employee`, instances of `HourlyEmployee`, `SalaryEmployee`, and `Manager` will all be comparable.

There is much more to object-oriented programming and design than we have discussed in ECE 150.

In future courses, notably ECE 155, you will learn more about object-oriented design.