Contents

Polymorphism

Motivation

Inheritance[1] provides another very useful mechanism: (subtype) *polymorphism*[2]. In a nutshell, the idea is that if a `Pyramid`[3] class extends the `Rectangle` class, then a `Pyramid` object can *still access all the `Rectangle`'s public methods, properties and attributes*. Indeed, a `Pyramid` *is a* `Rectangle`: this is precisely what polymorphism means.

While the example below is abstract, it can be easily instantiated to e.g., a `Cat` class inheriting from a `Pet` class or a `Pyramid` class inheriting from a `Rectangle` class.

Inheriting Attributes, Properties and Methods

Consider the following two classes:

```
class Class1
{
  private string attribute1;

  public void SetAttribute1(string aP)
  {
    attribute1 = aP;
  }

  public string Property1 { get; set; }
}

class Class2 : Class1
```

[1] https:/princomp.github.io/lectures/oop/inheritance

[2] https://en.wikipedia.org/wiki/Polymorphism_(computer_science)#Subtyping

[3] Technically, a "rectangular pyramid", if we require the pyramid to have a rectangle as its base.

```
{
  public string Property2 { get; set; }
}
```

Then,

- Any `Class1` object has an attribute `attribute1`, a property `Property1` and a method `SetAttribute1`.
- Any `Class2` object has the attribute, property and method of a `Class1` object, and *in addition*, it has a `Property2` property.

This means that the following code is valid:

```
class Program
{
  static void Main()
  {
    Class1 object1 = new Class1();
    object1.SetAttribute1("Test");
    object1.Property1 = "Test";

    Class2 object2 = new Class2();
    object2.SetAttribute1("Test");
    object2.Property1 = "Test";
    object2.Property2 = "Test";
  }
}
```

Note, however, that `object1.Property2 = "Test";` would not compile, since *an object from `Class1` cannot access the attributes, properties and methods of `Class2`*. Stated differently, an object in `Class2` *is a(n object in)* `Class1`, but the converse is not true: an object in `Class1` *is not* an object in `Class2`.

Polymorphism and References

Note that a `Class1` object can be created using a `Class2` constructor, since an object in `Class2` *is a(n object in)* `Class1`. Formally, we can write:

```
Class1 object3 = new Class2();
```

and then manipulate `object3` like any other *object from `Class1` (it is, in a way, "truncated"). In particular, we can use

```
object3.Property1 = "Test";
```

but `object3.Property2 = "Test";` would not compile *since we would be trying to access a property of `Class2` with a `Class1` object.*

Remember that an object in `Class1` is not* an object in `Class2`, and that the way we declared it, `object3` *is* a `Class1` object.

Solving Ambiguity by Overriding

For Methods

Now, consider the following class implementation and usage:

```
class Class1
{
  public string Test()
  {
    return "Class1";
  }
}

class Class2 : Class1
{
  public string Test()
  {
    return "Class2";
  }
}
```

```
using System;

class Program
{
  static void Main()
  {
    Class1 object1 = new Class1();
    Console.WriteLine(object1.Test());

    Class2 object2 = new Class2();
    Console.WriteLine(object2.Test());
  }
}
```

`Console.WriteLine(object1.Test());` will display "Class1": there is no ambiguity, since `object1` is a `Class1` object, it can access only the methods in its class.

However, the situation is less clear for `Console.WriteLine(object2.Test());`: since `object2` is "at the same time" a `Class1` and a `Class2` object, which method will be called? In this case, "Class2" will be displayed since C# prefers the "closest" method available (that is, the one in the same class as the calling object). However, a warning will be issued by the compiler because the `Test` method in `Class2` “hides” the inherited method `Test` from `Class1`.

A much better code explicitly instructs C# to *override* `Class1`'s `Test` method with `Class2`'s `Test` method. However, this further requires `Class1`'s `Test` method to explicitly give permission to be overriden, using the **`virtual`** keyword:

```
class Class1
{
  public virtual string Test()
  {
    return "Class1";
  }
}

class Class2 : Class1
{
  public sealed override string Test()
  {
    return "Class2";
  }
}

class Class3 : Class2
{
  public override string Test()
  {
    return "Class 3";
  }
}
```

This program will also display, as expected,

```
Class1
Class2
```

but this time the compiler will not complain: there is no ambiguity, as `Class2`'s `Test` method must explicitly take precedence when an object in `Class2` is calling a `Test` method.

Note that by default, methods are non-virtual, and non-virtual method cannot be overridden. However, overriding methods are treated as virtual and can be overridden themselves, unless they use the **`sealed`** keyword, as follows:

```
public override sealed string Test(){…}
```

Such a method **cannot** be overridden by classes inheriting from the class to which they belong.

Last but not least, note that an override method **must** have the same signature as the overridden method.

For Attributes and Properties

Virtual attributes and properties can similarly be overridden, provided of course the overriding property or attribute has the same datatype and name as the virtual method or property. Consider for instance an int Property in a Class1 class with no requirement that is inherited by a Class2 that wish to forbid negative values. One could do the following:

```
class Class1
{
  public virtual int Property { get; set; }
}
```

```
using System;

class Class2 : Class1
{
  private int attribute;
  public override int Property
  {
    set
    {
      if (value < 0)
        throw new ArgumentOutOfRangeException();
      else
        attribute = value;
    }
    get { return attribute; }
  }
}
```

Note that the property in Class2 has a backing field while there is no need for it in Class1.

The following would then throw an exception when the object2.Property = -12; statement would be executed:

```
using System;

class Program
{
  static void Main()
  {
    Class1 object1 = new Class1();
    object1.Property = -12;
    Class2 object2 = new Class2();
    try
    {
      object2.Property = -12;
    }
    catch
    {
      Console.WriteLine(
        "In Class2, Property cannot be set to a negative
         ↪  value."
      );
    }
  }
}
```

Note that, as for methods, overriding properties are by default virtual and can be overridden, for example as follows:

```
class Class3 : Class2
{
  public override int Property { set; get; }
}
```