Object-oriented programming concepts: Polymorphism and interfaces

by Michelle Yaiser

Adobe
michelleyaiser.com

Content

Created
6 February 2012

Page tools

Share on Facebook
Share on Twitter
Share on LinkedIn
Print

ActionScript inheritance
OOP

Requirements

Prerequisite knowledge
This article is designed for intermediate ActionScript developers. An understanding of ActionScript 3 language fundamentals, using inheritance, and using encapsulation in ActionScript 3 is required.

User level
Intermediate

Required products
Flash Builder (Download trial)
Flash Professional (Download trial)

Inheritance, encapsulation, abstraction and polymorphism are four of the fundamental concepts of object-oriented programming. You should have already learned about inheritance and encapsulation in previous articles. This article focuses on polymorphism, which requires an understanding of inheritance and encapsulation. Please read Object-oriented programming concepts: Inheritance and Object-oriented programming concepts: Encapsulation if you have not already.

Polymorphism is the concept that multiple types of objects might be able to work in a given situation. For example, if you needed to write a message on a piece of paper, you could use a pen, pencil, marker or even a crayon. You only require that the item you use can fit in your hand and can make a mark when pressed against the paper. Interfaces are a way to wrap all of those requirements into a single word (or phrase), for example: I need a writing instrument. This article will cover how to use polymorphism and interfaces in ActionScript 3.

Polymorphism through subclasses

There are multiple ways to achieve polymorphism when programming in ActionScript 3. The first is called *subclass polymorphism*, but technically, it is really using the concept of inheritance. As an example, look at this simple Pen class:

```
package writing {
    public class Pen {
        public function Pen() {}

        public function line():void {
            trace("Pen drew a line");
        }

        public function circle():void {
            trace("Pen drew a circle");
        }
    }
}
```

Then Pen class has two methods: one for drawing a line and one for drawing a circle. Using inheritance, you can create two new types of Pens: `FountainPen` and `RollerBallPen` . The `FountainPen` class extends the Pen class and adds one additional method named `refill()`.

```
package writing {
    public class FountainPen extends Pen {
        override public function line():void {
            trace("FountainPen drew a line... and a splotch");
        }

        override public function circle():void {
            trace("FountainPen almost drew a circle. Need more ink.");
```

```
        }

        public function refill():void {
            trace("all full");
        }
    }
}
```

The RollerBallPen also extends the Pen class.

```
package writing {
    public class RollerBallPen extends Pen {
        override public function line():void {
            trace("RollerBallPen drew a super smooth line.");
        }

        override public function circle():void {
            trace("RollerBallPen easily drew a circle.");
        }
    }
}
```

In this example, you can say that FountainPen **is** a Pen and RollerBallPen **is** a Pen . The fact that the compiler recognizes each of these classes as still being a Pen makes this type of polymorphism possible. Look at the following function:

```
function usePen( pen:Pen ):void {
    pen.line();
    pen.circle();
}
```

The function accepts a Pen instance and calls the `line()` function and the `circle()` function of that Pen instance. You could use the function in the following way:

```
var pen:Pen = new Pen();
usePen( pen );
```

You would receive this output:

```
Pen drew a line
Pen drew a circle
```

However, since FountainPen is also a type of Pen , you can pass it to the `usePen()` function. Look at both options in the code below:

```
var pen:Pen = new Pen();
usePen( pen );

var fountainPen:FountainPen = new FountainPen();
usePen( fountainPen );
```

You would receive this output:

```
Pen drew a line
Pen drew a circle

FountainPen drew a line... and a splotch
FountainPen almost drew a circle. Need more ink.
```

In both cases, the `usePen()` function was expecting a Pen instance. However, since both Pen and `FountainPen` are of type Pen , this function can accept either. In its most basic sense, this is subclass polymorphism. Pen or any of its subclasses can be used when the code requires a Pen , just as you would be willing to accept any type of pen from a colleague when you need to sign a document.

Subclass polymorphism is one technique that provides flexibility and allows future extension of your code base. Look at the code below. In it, each type of Pen is added to an array and the `usePen()` function is called.

```
var penArray:Array = [];
penArray [ 0 ] = new Pen();
penArray [ 1 ] = new FountainPen();
penArray [ 2 ] = new RollerBallPen();

for each ( var pen:Pen in penArray ) {
    usePen( pen );
}
```

As you may be expecting by now, it produces the following output:

```
Pen drew a line
Pen drew a circle
FountainPen drew a line... and a splotch
FountainPen almost drew a circle. Need more ink.
RollerBallPen drew a super smooth line.
RollerBallPen easily drew a circle.
```

The code sample shows that `FountainPen` and `RollerBallPen` can be used anywhere a Pen is used. It is important to understand that the inverse is not true. A Pen cannot be used anywhere a `FountainPen` is used. A `FountainPen` has the same methods as Pen plus an additional method named `refill()` . Therefore this code will fail to compile:

```
function useFountainPen( pen:FountainPen ):void {
    pen.line();
    pen.circle();
}
var pen:Pen = new Pen();
useFountainPen( pen ); //error
```

The error you will receive is:

```
Implicit coercion of a value with static type writing:Pen to a possibly unrelated
```

You receive this error because you are trying to treat a Pen as a `FountainPen` . Because it is missing the `refill()` method, Pen is not the same as `FountainPen` . With subclass polymorphism, you can only substitute subclasses for their super class. You cannot substitute a super class for one of its subclasses.

Subclass polymorphism limitations

Subclassing is one way to achieve polymorphism and it works for a number of cases; however, there are times when it is too inflexible. Continuing with the writing instrument example, what happens if you desperately need to draw a line and a circle but the only thing your artistic colleague has to offer is a hunk of charcoal? Could you still draw that line and circle? Of course, but does it make logical sense to say that charcoal is a type of Pen? Not really.

So, what do you do? Well, you could argue that `Charcoal` and Pen are both types of writing instruments and hence they both should actually descend from a common base class called `WritingInstrument` . This argument is actually a bit of a trap, but follow it through to understand why it may not work. Look at the following code:

```
package writing {
    public class WritingInstrument {
```

```
        public function WritingInstrument() {}
        public function line():void {
            trace("Writing Instrument drew a line");
        }

        public function circle():void {
            trace("Writing Instrument drew a circle");
        }
    }
}
package writing {
    public class Pen extends WritingInstrument {…}
}

package writing {
    public class Charcoal extends WritingInstrument {
        public function Charcoal(){}
        public function burn():void {
            trace("Charcoal now burning");
        }
    }
}
```

Initially, extending `WritingInstrument` seems fine. `Pen` and `Charcoal` are now both types of writing instruments. You could write a new function named `useWritingInstrument()` and you could use either a `Pen` or `Charcoal` to draw lines and circles.

```
function useWritingInstrument( instrument:WritingInstrument ):void {
    instrument.line();
    instrument.circle();
}
var pen:Pen = new Pen();
useWritingInstrument( pen );

var charcoal:Charcoal = new Charcoal();
useWritingInstrument( charcoal );
```

However, charcoal has other uses. For example, it can be used in a grill to cook. So, perhaps at some point you will write a function called `burnIt()` that accepts `Charcoal` :

```
function burnIt( charcoal:Charcoal ):void {
    charcoal.burn();
}
```

So far so good. You can write with your charcoal and you can burn it when you are done. However, as long as you are burning things, there are likely a number of other things around the office that could burn. What if you wanted to burn that paper you were drawing lines and circles upon?

```


package paper {
    public class Paper {
        public function burn():void {
            trace("Paper now burning");
        }
    }
}
```

How would you pass the paper to the `burnIt()` function? The function requires `Charcoal` presently. It doesn't make logical sense to have `Paper` extend `Charcoal` because that would mean that `Paper` is `Charcoal` . Additionally, because `Charcoal` is a `WritingInstrument` , it would mean that `Paper` is a `WritingInstrument` , which doesn't make any sense at all.

Hopefully you see that although `Paper` is not `Charcoal` , `Paper` and `Charcoal` can both burn. Further, `Charcoal` is not just a writing instrument, but `Charcoal` and `Pen` can both draw lines and

circles. Thus, using inheritance in this situation won't achieve the desired results. Instead, you need to use interfaces.

Public interfaces and contracts

The combination of all public methods and properties of an object form the *object's public interface*. In other words, the public methods and properties form a set of expectations that you can always count on. Interfaces define the required behavior for something to be considered a specific type of thing. They do not actually contain the required functionality. An interface is often called a *contract*.

You know that the `line()` and `circle()` methods exist as public methods in Pen . You can call them anytime in your code. By contract, being a Pen means always having those two methods available and public.

Consider the following reasoning:

- Pen 's public interface consists of two methods named `line()` and `circle()`
- `FountainPen` and `RollerBallPen` descend from Pen
- Because `FountainPen` and `RollerBallPen` descend from Pen , each has the two methods named `line()` and `circle()` in their public interfaces
- Therefore `FountainPen` and `RollerBallPen` can be used to fulfill the same contract as Pen

This reasoning is the basis of subclass polymorphism as discussed above. Continue with this contract example.

- The public `refill()` method was added to the `FountainPen` class
- `FountainPen` 's public interface now consists of the `line()` and `circle()` methods *and* the `refill()` method
- Only classes that have `line()` , `circle()` , and `refill()` methods in their public interfaces can be used to fulfill the same contract as `FountainPen`

Now you see why a super class cannot be substituted for one its subclasses—the super class does not fulfill the same contract as its subclass.

Defining and implementing interfaces

You've already determined that `Paper` , `Charcoal` , and Pen should not be related through inheritance but that both `Paper` and `Charcoal` should have the ability to burn and that both `Charcoal` and Pen should have the ability to draw lines and circles. To achieve that goal, define interfaces with those behaviors that those objects can fulfill.

Look at this example interface, `IWritingInstrument` :

```
package writing {
    public interface IWritingInstrument {
        function line():void;
        function circle():void;
    }
}
```

Because you are defining an interface, use the `interface` keyword followed by the name of the interface. By convention, interface names begin with an "I" so you can quickly distinguish them from classes. Note that the function definitions do not have access modifiers such as `public` or `private` . In a public interface, it is a requirement that the methods are public. Also note that the function definitions do not have function bodies because interfaces do not contain any functionality. This example interface is defining a contract that says in order to be considered an `IWritingInstrument` object, an object must have public `line()` and `circle()` methods that do not have parameters and return nothing.

Interfaces are not classes. Therefore, you can't instantiate an interface. You will receive an error if you try to instantiate an interface as you can see in the following code:

```
var iWriting:IWritingInstrument = new IWritingInstrument(); //error
```

To apply an interface to a class, use the `implements` keyword followed by the name of the interface in your class declaration. Below, the Pen class is changed to implement the `IWritingInstrument` interface.

```
package writing {
    public class Pen implements IWritingInstrument {

        public function Pen(){}

        public function line():void {
            trace("Pen drew a line");
        }

        public function circle():void {
            trace("Pen drew a circle");
        }
    }
}
```

You can see that the Pen class now fulfills the contract required to be an `IWritingInstrument` . As with inheritance, you can say that Pen **is** an `IWritingInstrument` . Further, because `FountainPen` and `RollerBallPen` inherit from Pen , it is proper to say that both of those classes are of type `IWritingInstrument` .

You can also use an interface as a type for properties or parameters. In this example, the `useWritingInstrument()` method can accept any object that implements the `IWritingInstrument` and call its `line()` and `circle()` methods.

```
function useWritingInstrument( instrument:IWritingInstrument ):void {
    instrument.line();
    instrument.circle();
}
var pen:Pen = new Pen();
useWritingInstrument( pen );
var fountainPen:FountainPen = new FountainPen();
useWritingInstrument( fountainPen );
```

The `Charcoal` class should also implement the `IWritingInstrument` interface:

```
package writing {
    public class Charcoal implements IWritingInstrument {

        public function Charcoal(){}

        public function line():void {
            trace("Charcoal drew a thick line");
        }

        public function circle():void {
            trace("Charcoal drew an approximate circle");
        }

        public function burn():void {
            trace("Charcoal now burning");
        }
    }
}
```

`Charcoal` is now a valid type of `IWritingInstrument` . Hence, you can also pass it to the `useWritingInstrument()` function.

```
function useWritingInstrument( instrument:IWritingInstrument ):void {
    instrument.line();
    instrument.circle();
}
var pen:Pen = new Pen();
useWritingInstrument( pen );
var fountainPen:FountainPen = new FountainPen();
useWritingInstrument( fountainPen );
```

```
var charcoal:Charcoal = new Charcoal();
useWritingInstrument( charcoal );
```

The `useWritingInstrument()` function does not care if you give it a `Pen` , `Charcoal` or any other specific implementation. Instead, it only cares that you provide it something that fulfills the correct contract—the `IWritingInstrument` contract. Using an interface in this way is a very useful technique, often referred to a *programming to an interface instead of an implementation*.

You can also define an interface for things that are combustion sources:

```
package fire {
    public interface ICombustionSource {
        function burn():void;
    }
}
```

Although you can only inherit from one super class, you can implement as many interfaces as you need. For example, `Charcoal` is both a writing instrument and a combustion source:

```
package writing {
    import fire.ICombustionSource;

    public class Charcoal implements IWritingInstrument, ICombustionSource {
        public function Charcoal(){}
        public function line():void {
            trace("Charcoal drew a thick line");
        }

        public function circle():void {
            trace("Charcoal drew an approximate circle");
        }

        public function burn():void {
            trace("Charcoal now burning");
        }
    }
}
```

Once you change the `burnIt()` function to accept an `ICombustionSource` , a charcoal object can be used both to write and to burn:

```
function useWritingInstrument( instrument:IWritingInstrument ):void {
    instrument.line();
    instrument.circle();
}

function burnIt( source:ICombustionSource ):void {
    source.burn();
}
var pen:Pen = new Pen();
useWritingInstrument( pen );

var charcoal:Charcoal = new Charcoal();
useWritingInstrument( charcoal );
burnIt( charcoal );
```

Finally, `Paper` should also have the ability to burn. Change the `Paper` class to implement the `ICombustionSource` interface then it too can burn.

```
package paper {
    import fire.ICombustionSource;

    public class Paper implements ICombustionSource {
        public function Paper(){}
```

```
        public function burn():void {
            trace("Paper now burning");
        }
    }
}
var pen:Pen = new Pen();
useWritingInstrument( pen );

var charcoal:Charcoal = new Charcoal();
useWritingInstrument( charcoal );
burnIt( charcoal );

var paper:Paper = new Paper();
burnIt( paper );
```

As you can see, Pen and `Charcoal` objects are now available for writing. The `Charcoal` and `Paper` objects are available for burning. Yet, `Charcoal` , Paper and Pen do not share a common super class. If you ran the code directly above, you would receive the following output:

```
Pen drew a line
Pen drew a circle
Charcoal drew a thick line
Charcoal drew an approximate circle
Charcoal now burning
Paper now burning
```

Where to go from here

Programming to interfaces instead of implementations provides for future expansion as well as flexibility in the way you construct the objects in your applications. Further, it forces you to consider the actual function of each of your classes. These techniques form the basis of the largest and most maintainable applications.

More Like This

Object-oriented programming concepts: Encapsulation
Object-oriented programming concepts: Inheritance
Object-oriented programming concepts: Writing classes
Object-oriented programming concepts: Objects and classes
Object-oriented programming concepts: Composition and aggregation