

# POLYMORPHISM - THE BIG PICTURE\*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNX and licensed under the  
Creative Commons Attribution License 3.0<sup>†</sup>

## Abstract

Learn the essence of runtime polymorphism.

NOTE: Click Polymorph02 <sup>1</sup> to run this ActionScript program. (Click the "Back" button in your browser to return to this page.)

## 1 Table of Contents

- Preface (p. 2)
  - General (p. 2)
  - Viewing tip (p. 2)
    - \* Figures (p. 2)
    - \* Listings (p. 3)
  - Supplemental material (p. 3)
- General background information (p. 3)
- Preview (p. 5)
- Discussion and sample code (p. 10)
  - The file named Polymorph02.mxml (p. 10)
  - The file named Driver.as (p. 11)
  - The file named MyShape.as (p. 13)
  - The file named MyCircle.as (p. 14)
  - The file named MyRectangle.as (p. 15)
- Run the program (p. 15)
- Resources (p. 15)
- Complete program listing (p. 15)
- Miscellaneous (p. 18)

---

\*Version 1.3: May 19, 2010 11:50 pm +0000

<sup>†</sup><http://creativecommons.org/licenses/by/3.0/>

<sup>1</sup><http://cnx.org/content/m34447/latest/Polymorph02.html>

## 2 Preface

### 2.1 General

NOTE: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming (*OOP*) with ActionScript.

#### **The three main characteristics of an object-oriented program**

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I explained both encapsulation and inheritance from a big-picture viewpoint in previous lessons. (See *Baldwin's ActionScript programming website* <sup>2</sup> .) There are two different ways to implement polymorphism:

- Polymorphism based on class inheritance
- Polymorphism based on interface inheritance

I will explain and illustrate polymorphism based on class inheritance in this lesson and will explain and illustrate polymorphism based on interface inheritance in the next lesson.

#### **Several ways to create and launch ActionScript programs**

There are several ways to create and launch programs written in the ActionScript programming language. Most of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. (See *Baldwin's Flex programming website* <sup>3</sup> .) You should study that lesson before embarking on the lessons in this series.

#### **Some understanding of Flex MXML will be required**

I also recommend that you study all of the lessons on Baldwin's Flex programming website <sup>4</sup> in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

#### **Will emphasize ActionScript code**

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

### 2.2 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

#### 2.2.1 Figures

- Figure 1 (p. 5) . File structure for Polymorph02.
- Figure 2 (p. 6) . Program output at startup.
- Figure 3 (p. 7) . Sample output from area method of a MyShape object.
- Figure 4 (p. 8) . Sample output from area method of a MyCircle object.
- Figure 5 (p. 9) . Sample output from area method of a MyRectangle object.

---

<sup>2</sup><http://www.dickbaldwin.com/tocActionScript.htm>

<sup>3</sup><http://www.dickbaldwin.com/tocFlex.htm>

<sup>4</sup><http://www.dickbaldwin.com/tocFlex.htm>

### 2.2.2 Listings

- Listing 1 (p. 10) . Source code for the file named Polymorph02.mxml.
- Listing 2 (p. 11) . Beginning of the class named Driver.
- Listing 3 (p. 11) . The constructor for the class named Driver.
- Listing 4 (p. 12) . Beginning of the method named buttonHandler.
- Listing 5 (p. 13) . Instantiate an object.
- Listing 6 (p. 13) . Call the area method on the object.
- Listing 7 (p. 13) . The class named MyShape.
- Listing 8 (p. 14) . The class named MyCircle.
- Listing 9 (p. 15) . Listing of the file named Polymorph02.mxml.
- Listing 10 (p. 16) . Listing of the file named Driver.as.
- Listing 11 (p. 17) . Listing of the file named MyShape.as.
- Listing 12 (p. 17) . Listing of the file named MyCircle.as.
- Listing 13 (p. 17) . Listing of the file named MyRectangle.as

### 2.3 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at [www.DickBaldwin.com](http://www.DickBaldwin.com) <sup>5</sup> .

## 3 General background information

The first of three major characteristics of an object-oriented program is encapsulation, which I explained in a previous lesson. The second of the three major characteristics is inheritance, and the third is polymorphism. I also explained inheritance in a previous lesson. I will explain polymorphism from a big-picture viewpoint in this lesson.

### Not as complicated as it sounds

Upon first reading, you may conclude that polymorphism sounds extremely complicated. However, polymorphism is more difficult to explain than it is to program. Once you have read the description and have seen the concept applied to an actual program, you will (*hopefully*) conclude that it is not as complicated as it sounds.

### What is polymorphism?

Polymorphism is a word taken from the Greek, meaning "many forms", or words to that effect.

The purpose of polymorphism as it applies to OOP is to allow one name to be used to specify a general class of actions. Within that general class of actions, the specific action that is applied in any particular situation is determined by the type of data involved.

Polymorphism in ActionScript comes into play when inherited methods are overridden to cause them to behave differently for different types of subclass objects.

### Overriding versus overloading methods

If you read much in the area of OOP, you will find the words *override* and *overload* used frequently. (*This lesson deals with overriding methods and does not deal with overloading.*)

Some programming languages such as C++, Java, and C# support a concept known as method or constructor overloading. However, ActionScript 3 does not support method or constructor overloading. Overriding a method is an entirely different thing from overloading a method even for those languages that support overloading.

### Modify the behavior of an inherited method

Polymorphism can exist when a subclass modifies or customizes the behavior of a method inherited from its superclass in order to meet the special requirements of objects instantiated from the subclass. This is known as *overriding* a method and requires the use of the **override** keyword in ActionScript.

---

<sup>5</sup><http://www.dickbaldwin.com/toc.htm>

### Override methods differently for different subclasses

ActionScript supports the notion of *overriding* a method inherited from a superclass to cause the named method to behave differently when called on objects of different subclasses, each of which extends the same superclass and overrides the same method name.

#### Example - compute the area of different geometrical shapes

For example, consider the computation of the area of a geometrical shape in a situation where the type of geometrical shape is not known when the program is compiled. Polymorphism is a tool that can be used to handle this situation.

#### Circle and Rectangle extend Shape

Assume that classes named **Circle** and **Rectangle** each extend a class named **Shape**. Assume that the **Shape** class defines a method named **area**. Assume further that the **area** method is properly overridden in the **Circle** and **Rectangle** classes to return the correct area for a circle or a rectangle respectively.

#### Three types of objects

In this case, a **Circle** object is a **Shape** object because the **Circle** class extends the **Shape** class. Similarly, a **Rectangle** object is also a **Shape** object.

Therefore, an object of the **Shape** class, the **Circle** class, or the **Rectangle** class can be instantiated and any one of the three can be saved in a variable of type **Shape**.

#### Flip a virtual random coin

Assume that the program flips a virtual coin and, depending on the outcome of the flip, instantiates an object of either the **Circle** class or the **Rectangle** class and saves it in a variable of type **Shape**. Assuming that the coin flip is truly random, the compiler cannot possibly know at compile time which type of object will be stored in the variable at runtime.

#### Two versions of the area method

Regardless of which type of object is stored in the variable, the object will contain two versions of the method named **area**. One version is the version that is defined in the **Shape** class, and this version will be the same regardless of whether the object is a circle or a rectangle.

Also, this version can't return a valid area because a general shape doesn't have a valid area. However, if the area method is defined to return a value, even this version must return a value even if it isn't valid. (*Other programming languages get around this problem with something called an abstract class, which isn't allowed in ActionScript 3.*)

The other version of the **area** method will be different for a **Circle** object and a **Rectangle** object due simply to the fact the algorithm for computing the area of a circle is different from the algorithm for computing the area of a rectangle.

#### Call the area method on the object

If the program calls the **area** method on the object stored in the variable of type **Shape** at runtime, the correct version of the **area** method will be selected and executed and will return the correct area for the type of object stored in that variable. This is *runtime polymorphism* based on method overriding.

#### A more general description of runtime polymorphism

A reference variable of a superclass type can be used to reference an object instantiated from any subclass of the superclass.

If an overridden method in a subclass object is called using a superclass-type reference variable, the system will determine, at runtime, which version of the method to use based on the true type of the object, and not based on the type of reference variable used to call the method.

#### The general rule

The type of the reference determines the names of the methods that can be called on the object. The actual type of the object determines which of possibly several methods having the same name will be executed.

#### Selection at runtime

Therefore, it is possible (*at runtime*) to select among a family of overridden methods and determine which method to execute based on the type of the subclass object pointed to by the superclass-type reference when the overridden method is called on the superclass-type reference.

### Runtime polymorphism

In some situations, it is possible to identify and call an overridden method at runtime that cannot be identified at compile time. In those situations, the identification of the required method cannot be made until the program is actually running. This is often referred to as *late binding*, *dynamic binding*, or *run-time polymorphism*.

## 4 Preview

### Project file structure

Figure 1 shows the project file structure for the Flex project named **Polymorph02**.

File structure for Polymorph02.

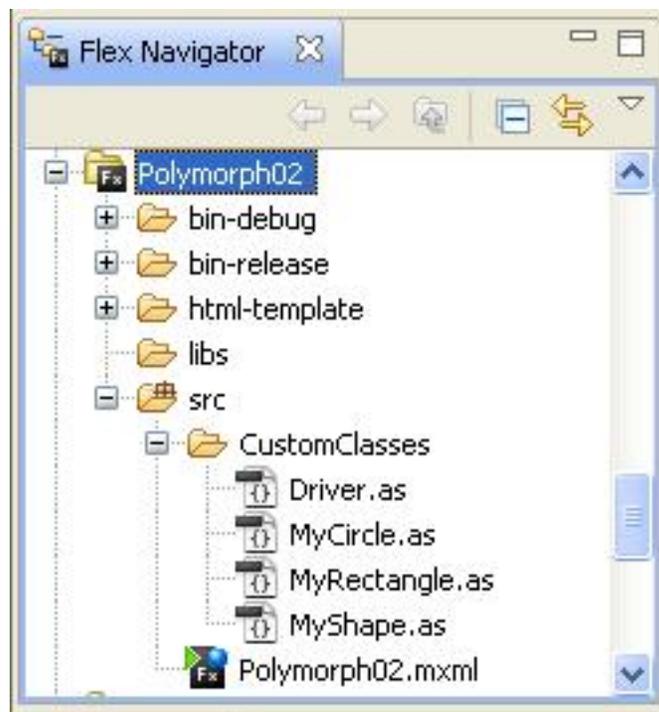


Figure 1: File structure for Polymorph02.

The image in Figure 1 was captured from the Flex Navigator panel of Flex Builder 3.

### Program output at startup

Figure 2 shows the program output at startup.

---

Program output at startup.

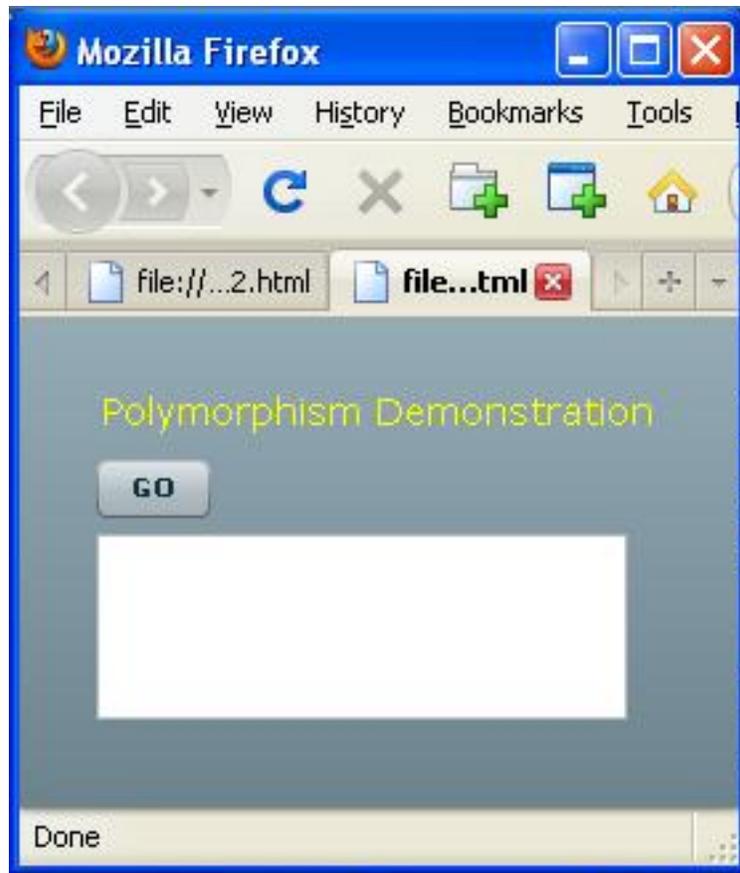


Figure 2: Program output at startup.

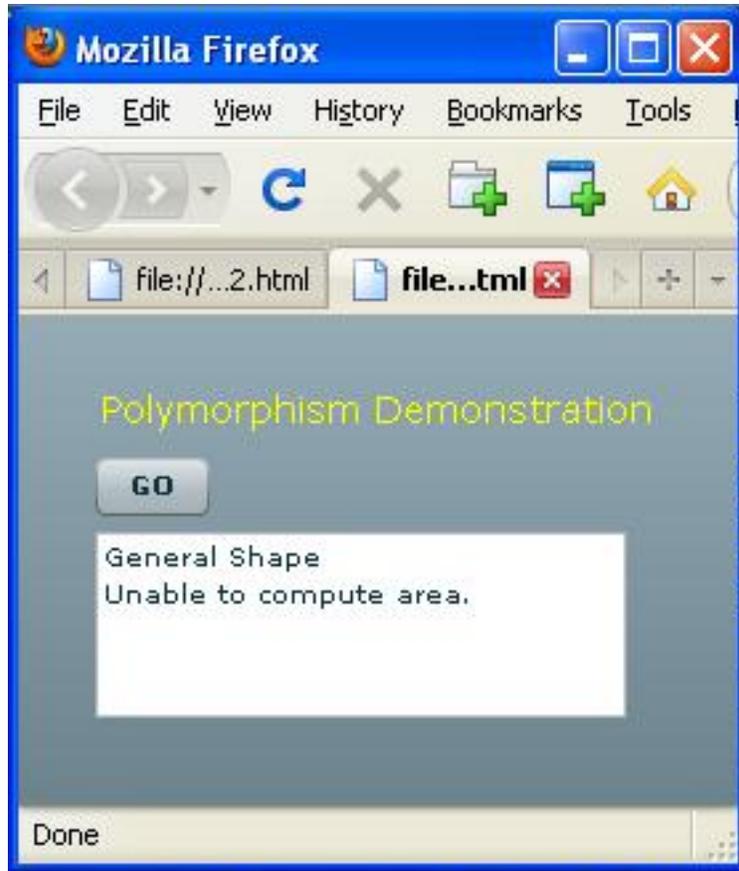
---

### The program GUI

As you can see, the program GUI consists of a label, a button and an empty text area. Each time the user clicks the button, an object is instantiated, and information about that object (*similar to the information shown in Figure 3*) is displayed in the text area.

---

Sample output from area method of a MyShape object.



**Figure 3:** Sample output from area method of a MyShape object.

---

### Program class definitions

This program defines the following four classes as shown in the folder named **CustomClasses** in Figure 1:

- **Driver**
- **MyCircle**
- **MyRectangle**
- **MyShape**

As the name implies, the **Driver** class is the driver for the entire program. For example, it creates the GUI shown in Figure 2 at startup and updates the GUI each time the user clicks the **GO** button as shown in Figure 3.

The classes named **MyCircle** and **MyRectangle** each extend the class named **MyShape**. Therefore, specialized shape objects can be instantiated from these two classes.

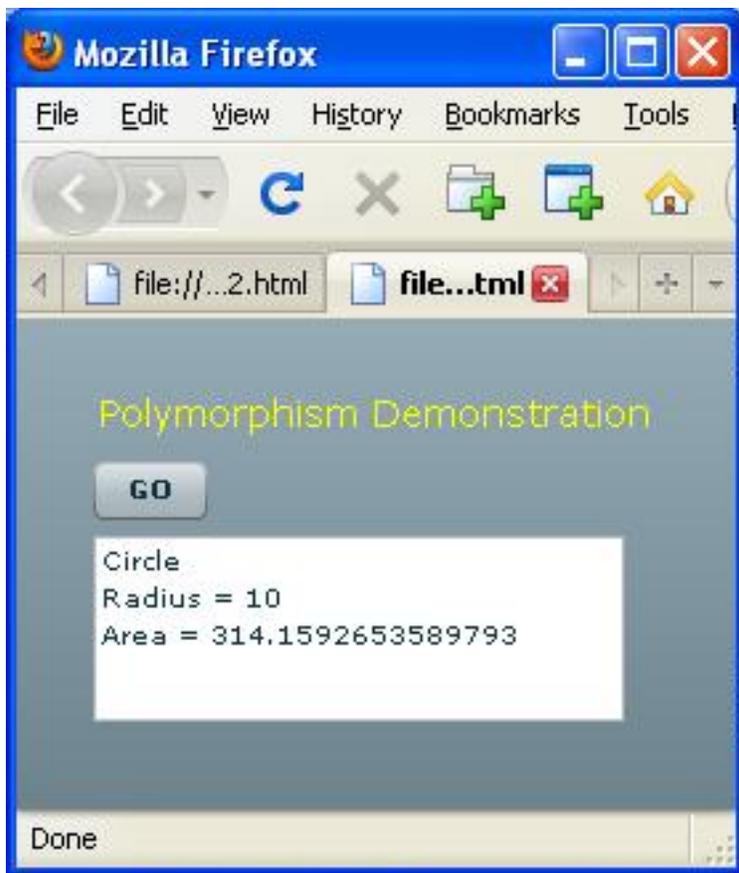
The class named **MyShape** defines an instance method named **area**, which returns the text shown in Figure 3 each time it is executed.

**Sample output from area method of a MyCircle object**

The class named **MyCircle** overrides the inherited **area** method to return a text string similar to that shown in Figure 4 each time it is executed.

---

**Sample output from area method of a MyCircle object.**



**Figure 4:** Sample output from area method of a MyCircle object.

---

**The radius is a random value**

Note, however, that the value of radius is established from a random number generator each time an object of the **MyCircle** class is instantiated, so the actual values for *Radius* and *Area* in Figure 4 will change each time the button is clicked.

**Sample output from area method of a MyRectangle object**

The class named **MyRectangle** also overrides the inherited **area** method to return a text string similar to that shown in Figure 5 each time it is executed.

---

Sample output from area method of a MyRectangle object.

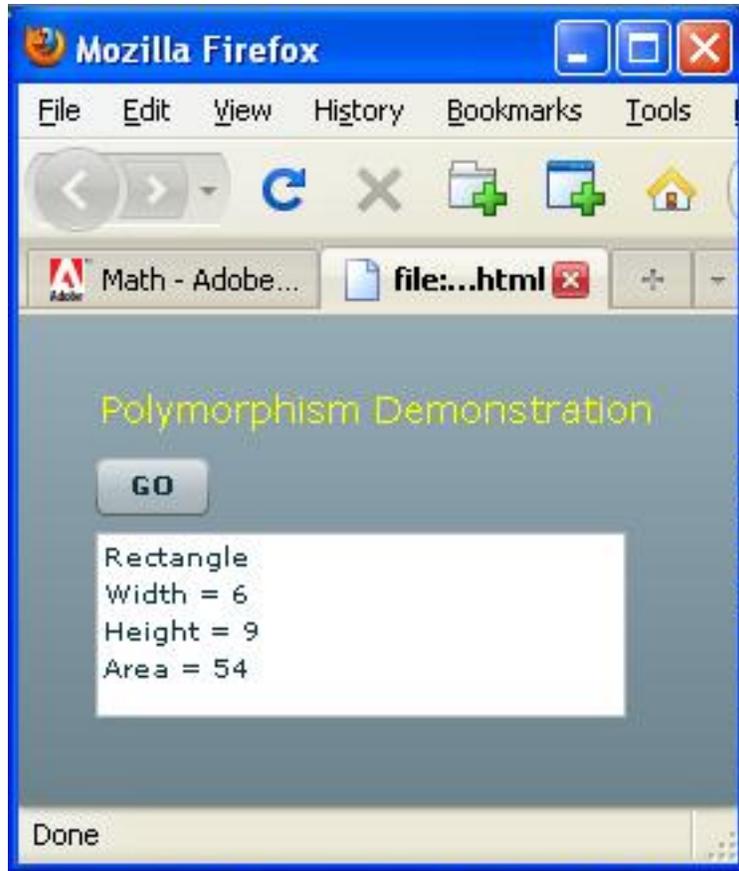


Figure 5: Sample output from area method of a MyRectangle object.

---

#### Width and height are random values

Once again, however, the values of Width and Height are established from a random number generator each time an object of the **MyRectangle** class is instantiated, so the actual values for *Width*, *Height*, and *Area* in Figure 5 will change each time the button is clicked.

#### A click event handler

A **click** event handler is registered on the button shown in Figure 5. Each time the button is clicked, the event handler uses the output from a random number generator to instantiate an object of one of the **following classes** and saves it in a variable of type **MyShape**.

- MyShape
- MyCircle
- MyRectangle

#### Multiple versions of the method named area

As I explained earlier (p. 4) , an object of the **MyShape** class will contain one version of the **area** method, but objects of the other two classes will each contain two versions of the **area** method.

One version of the method is common among all three objects, and that is the version that returns the text shown in Figure 3.

However, the other version in each of the **MyCircle** and **MyRectangle** classes is unique to its class returning values similar to those shown in Figure 4 and Figure 5.

#### **No selection requirement for object of type MyShape**

When the **area** method is called on an object of the **MyShape** class, there is no requirement to select a "correct" version of the method because that object only contains one version of the method.

#### **Polymorphism kicks in...**

Because the **area** method is defined in the **MyShape** class and overridden in the **MyCircle** class and the **MyRectangle** class, and because the objects of the **MyCircle** class and the **MyRectangle** class are saved as type **MyShape** , polymorphism kicks in when the **area** method is called on those objects. The overridden method is selected for execution in those cases where the object contains an overridden version of the method.

#### **Could easily expand the shape system**

Because of polymorphism, I could easily expand this system of shapes by defining new classes for new shapes (*such as triangle, octagon, hexagon, etc.*) , without any requirement to modify the **MyShape** class, the **MyCircle** class, or the **MyRectangle** class.

Assuming that I cause the classes for the new shapes to extend the **MyShape** class and properly override the **area** method to return the correct values, I could then instantiate objects of the new classes, save them as type **MyShape** , call the **area** method on those new objects and expect the correct version of the **area** method to be executed.

#### **The most powerful concept...**

This is runtime polymorphism in all its glory, and is probably the most powerful concept in all of OOP.

## **5 Discussion and sample code**

### **Will discuss in fragments**

I will discuss the code in the five files shown in Figure 1 in fragments. Complete listings of those files are provided in Listing 9 through Listing 13 near the end of the lesson.

### **5.1 The file named Polymorph02.mxml**

The complete MXML code for the file named **Polymorph02.mxml** is provided in Listing 1 and also in Listing 9.

#### **Listing 1: Source code for the file named Polymorph02.mxml.**

```
<?xml version="1.0" encoding="utf-8"?>

<!--Illustrates polymorphism.-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

In keeping with my objective of emphasizing ActionScript programming rather than Flex MXML programming in this series of tutorial lessons, this program is written almost entirely in ActionScript. The MXML file serves simply as a launch pad for the program by instantiating an object of the class named **Driver** .

## 5.2 The file named Driver.as

A complete listing of this file is provided in Listing 10 near the end of the lesson. The class named **Driver** begins in Listing 2.

### Listing 2: Beginning of the class named Driver.

```
package CustomClasses{
import flash.events.*;

import mx.containers.VBox;
import mx.controls.Button;
import mx.controls.Label;
import mx.controls.TextArea;

public class Driver extends VBox{
    private var textArea:TextArea = new TextArea();
```

### A Driver object is a VBox object

The class named **Driver** extends the **VBox** class. As a result, components added to the objects are arranged vertically with left justification as shown in Figure 2.

### The TextArea object

Listing 2 instantiates the **TextArea** object shown in Figure 2 and saves its reference in an instance variable named **textArea** . The reference was saved in an instance variable to make it available to both the constructor and a **click** event handler method later.

### The constructor for the class named Driver

The constructor for the class named **Driver** is shown in Listing 3.

### Listing 3: The constructor for the class named Driver.

```
    public function Driver(){//constructor
var label:Label = new Label();
label.text = "Polymorphism Demonstration";
label.setStyle("fontSize",14);
label.setStyle("color",0xFFFF00);
addChild(label);

var button:Button = new Button();
button.label = "GO";
addChild(button);

textArea.width = 200;
textArea.height = 70;
addChild(textArea);
```

```

        button.addEventListener(
            MouseEvent.CLICK,buttonHandler);
    }//end constructor

```

### Nothing new here

I don't believe there is anything in Listing 3 that you haven't seen in previous lessons. The code in the constructor simply constructs the GUI pictured in Figure 2.

### A click event handler method

Probably the most important thing to note about listing 3 is the registration of the **click** event handler method named **buttonHandler** on the button shown in Figure 2. Once the GUI is constructed, further action occurs only when the user clicks the button, causing the method named **buttonHandler** to be executed.

### Beginning of the method named buttonHandler

The method named **buttonHandler** begins in Listing 4. This method is executed each time the user clicks the button shown in Figure 2.

### Listing 4: Beginning of the method named buttonHandler.

```

    private function buttonHandler(
        event:MouseEvent):void{
    var randomChoice:Number = Math.random();

    var radius:uint = uint(10*Math.random() + 1);
    var width:uint = uint(10*Math.random() + 1);
    var height:uint = uint(10*Math.random() + 1);

```

### Four local variables

The most interesting thing in Listing 4 is the declaration and population of the following four local variables:

- **randomChoice**
- **radius**
- **width**
- **height**

### A random value

The variable named **randomChoice** is of type **Number** and is populated with the return value from a call to the **random** method of the **Math** class. The documentation tells us that this method:

*"Returns a pseudo-random number n, where 0 LTE n LT 1. The number returned is calculated in an undisclosed manner, and pseudo-random because the calculation inevitably contains some element of non-randomness."*

Note that LTE and LT represent "less than or equal" and "less than" respectively. (Lots of problems arise from including angle brackets and ampersands in html text so I avoid doing that when I can.)

### A fractional random value

Thus, the return value is a fractional value that is greater than or equal to zero and less than 1.0. This value will be used later to decide which type of object to instantiate.

### Three more random values

The last three variables in the above list (p. 12) are also populated with random values, but not with fractional values in the range from 0 to 0.999... Instead, these variables are populated with unsigned integer

values in the range of 1 to 10 inclusive. This is accomplished by multiplying the original random value by 10, adding 1 to that product, and casting the result to an unsigned integer.

#### A local variable of type `MyShape`

Listing 5 begins by declaring a variable of type `MyShape` that will be used to hold a reference to an object instantiated from one of the following classes: `MyShape`, `MyCircle`, or `MyRectangle`.

#### Listing 5: Instantiate an object.

```
var myShape:MyShape;

if(randomChoice < 0.33333){
    myShape = new MyShape();
}else if(randomChoice < 0.66666){
    myShape = new MyCircle(radius);
}else{
    myShape = new MyRectangle(width,height);
} //end else
```

#### Instantiate an object

Then Listing 5 uses the random value stored in the variable named `randomChoice` to make a decision and to instantiate an object of one of the three classes listed above (p. 9). The decision as to which type of object to instantiate and store in the variable named `myShape` is completely random and completely unknown to the compiler when the program is compiled.

#### Random values for radius, width, and height

Note also that Listing 5 also uses the other values stored in the other variables in the above list (p. 12) to specify the radius of the circle, or to specify the width and the height of the rectangle.

#### Call the area method on the object

Finally, (and this is the essence of runtime polymorphism), Listing 6 calls the `area` method on the randomly instantiated object and writes the return value into the text area shown in Figure 2.

#### Listing 6: Call the area method on the object.

```
        textArea.text = myShape.area();
    } //end buttonHandler

} //end class
} //end package
```

#### And the result will be...

The result will be similar to Figure 3, Figure 4, or Figure 5, depending on which type of object is instantiated and depending on the random values passed for radius, width, and height.

#### The end of the class

Listing 6 also signals the end of the class named `Driver`.

### 5.3 The file named `MyShape.as`

The class named `MyShape` is shown in its entirety in Listing 7, and also in Listing 11 near the end of the lesson.

#### Listing 7: The class named `MyShape`.

```

package CustomClasses{
public class MyShape{

    public function area():String{
        return "General Shape\n" +
            "Unable to compute area.";
    }// end area method

} //end class
} //end package

```

### The sole purpose

The sole purpose of the class named **MyShape** is to serve as the root of a hierarchy of more specialized shape classes and to provide a default version of the **area** method that can be overridden in the subclasses.

The class accomplishes those things well, and beyond that, there isn't much to say about the code in Listing 7. Note that the class doesn't even define a constructor but instead uses the default constructor that is provided by the compiler.

## 5.4 The file named MyCircle.as

The class named **MyCircle** is shown in its entirety in Listing 8, and also in Listing 12 near the end of the lesson.

### Listing 8: The class named MyCircle.

```

package CustomClasses{
public class MyCircle extends MyShape{
    private var radius:Number;

    public function MyCircle(radius:Number){ //constructor
        this.radius = radius;
    } //end constructor

    override public function area():String{
        return "Circle\n" +
            "Radius = " + radius + "\n" +
            "Area = " + Math.PI * radius * radius;
    } //end area
} //end class
} //end package

```

The class named **MyCircle** is only slightly more complicated than the class named **MyShape**.

### The constructor

The constructor for the class named **MyCircle** receives an incoming value for the radius and saves that value in a private instance variable named **radius**. Saving the value in an instance variable makes it available to the **area** method that will be executed later.

### Type coercion

Recall from Listing 4 and Listing 5 that the value that is actually passed to the constructor is of type **uint**, which is an unsigned integer. That value is coerced to type **Number** by passing it as an argument that is defined to be of type **Number**.

### An overridden area method

Note that the method named **area** is declared to be an **override**. This syntax is required by ActionScript when a method in a subclass overrides an inherited method.

#### Return a String object

The code in the **area** method concatenates several individual strings (*including the computed area of the circle*) into a single **String** object and returns a reference to that object. A sample of the returned value is shown displayed in the text area of Figure 4.

#### Concatenation of strings with numeric values

The computed value of the area and the stored value of the radius are both numeric values. When a numeric value is concatenated with a string, the numeric value is coerced into a string of characters, decimal points, etc., and the two strings are concatenated into a single string.

#### Computation of the area

As you can see in Listing 8, the area of the circle is computed as the square of the radius multiplied by the constant PI. (*Hopefully you recall that formula from your high school geometry class.*) As mentioned earlier, the resulting area value is concatenated with the string to its left where it becomes part of the larger **String** object that is returned by the method.

### 5.5 The file named MyRectangle.as

The class named **MyRectangle** is so similar to the class named **MyCircle** that it doesn't warrant a detailed explanation. A complete listing of the file is provided in Listing 13 near the end of the lesson.

The constructor receives and saves numeric values for the width and the height of the rectangle.

The overridden **area** method computes the area as the product of the width and the height and concatenates that information into a returned **String** object where it is displayed in the format shown in Figure 5.

## 6 Run the program

I encourage you to run <sup>6</sup> this program from the web. Then copy the code from Listing 9 through Listing 13. Use that code to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

## 7 Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

## 8 Complete program listing

Complete listings of the Flex and ActionScript files used in this program are provided in Listing 9 through Listing 13 below.

#### Listing 9: Listing of the file named Polymorph02.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--Illustrates polymorphism.-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
```

<sup>6</sup><http://cnx.org/content/m34447/latest/Polymorph02.html>

```
xmlns:cc="CustomClasses.*">
<cc:Driver/>
</mx:Application>
```

**Listing 10: Listing of the file named Driver.as.**

```
package CustomClasses{
import flash.events.*;

import mx.containers.VBox;
import mx.controls.Button;
import mx.controls.Label;
import mx.controls.TextArea;

public class Driver extends VBox{
    private var textArea:TextArea = new TextArea();

    public function Driver(){//constructor
        var label:Label = new Label();
        label.text = "Polymorphism Demonstration";
        label.setStyle("fontSize",14);
        label.setStyle("color",0xFFFF00);
        addChild(label);

        var button:Button = new Button();
        button.label = "GO";
        addChild(button);

        textArea.width = 200;
        textArea.height = 70;
        addChild(textArea);

        button.addEventListener(
            MouseEvent.CLICK,buttonHandler);
    }//end constructor

    private function buttonHandler(
        event:MouseEvent):void{
        var randomChoice:Number = Math.random();
        var radius:uint = uint(10*Math.random() + 1);
        var width:uint = uint(10*Math.random() + 1);
        var height:uint = uint(10*Math.random() + 1);
        var myShape:MyShape;

        if(randomChoice < 0.33333){
            myShape = new MyShape();
```

```

        }else if(randomChoice < 0.66666){
            myShape = new MyCircle(radius);
        }else{
            myShape = new MyRectangle(width,height);
        }
        textArea.text = myShape.area();
    }//end buttonHandler

} //end class
} //end package

```

**Listing 11: Listing of the file named MyShape.as.**

```

package CustomClasses{
public class MyShape{

    public function area():String{
        return "General Shape\n" +
            "Unable to compute area.";
    } // end area method
} //end class

} //end package

```

**Listing 12: Listing of the file named MyCircle.as.**

```

package CustomClasses{
public class MyCircle extends MyShape{
    private var radius:Number;

    public function MyCircle(radius:Number){ //constructor
        this.radius = radius;
    } //end constructor

    override public function area():String{
        return "Circle\n" +
            "Radius = " + radius + "\n" +
            "Area = " + Math.PI * radius * radius;
    } //end area
} //end class
} //end package

```

**Listing 13: Listing of the file named MyRectangle.as**

```

package CustomClasses{
public class MyRectangle extends MyShape{

```

```
private var width:Number;
private var height:Number;

public function MyRectangle(
    width:Number,height:Number){//constructor
    this.width = width;
    this.height = height;
} //end constructor

override public function area():String{
    return "Rectangle\n" +
        "Width = " + width + "\n" +
        "Height = " + height + "\n" +
        "Area = " + width * height;
} //end area
} //end class
} //end package
```

## 9 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Polymorphism - The Big Picture
- Files:
  - ActionScript0110\ActionScript0110.htm
  - ActionScript0110\Connexions\ActionScriptXhtml0110.htm

NOTE: **PDF disclaimer:** Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

-end-