Flex in a Week, Flex 4.5
Video 1.14: Implementing object-oriented programming concepts in Flex

I have been throwing around object-object programming terms like classes, properties and instances, throughout this first day of training.

In this video, I will formally introduce you to these terms and then you will learn how OOP applies to the MXML tags that you are already using.

You will also learn how to create your own MXML custom components with custom class properties.

A core concept behind object-oriented programming is that problems are easier to solve if you model your programming needs on the real world.

In OOP, your code describes the classes of objects involved in a problem to be solved and then allows those objects to interact as they would in the real world.

In this video I will introduce you to OOP terminology and discuss how classes, objects, properties, methods and other OOP concepts are implemented in the Flex framework.

OOP revolves around categories of objects, called classes.

A class is essentially a blueprint that can be used as a template for creating specified objects.

A property describes a characteristic of the class.

In programming terms, you would call this a variable that is assigned to the class.

A method describes the behaviors or actions available to instances of the class.

In programming terms, you would call this a function assigned to the class.

In OOP terminology, properties and methods are also known as class members.

This is a simple, real-world example of a class that is a blueprint for a category named Ball.

Every time the class blueprint is used to create an instance, the new ball will have properties like shape, material and color that define it.

Every ball will also have access to methods that define the actions and behaviors of the ball.

You can see that the Ball class has methods that define the way a ball can be thrown, caught or bounced.

Note that the word Ball itself is capitalized. By common convention, class names are capitalized while properties and methods use camel case.

Every ball that is created is called an instance of the Ball class.

You may also hear the term object used interchangeably with the term instance.

You can create one or more instances of a class and each instance can have unique property values and can behave in ways defined in the methods.

Here are three instances of the Ball class.

The first ball, which I have named ball1, is round, made of rubber, and is blue.

The second ball, ball2, is oval, made of leather and is brown.

The last ball, ball3, is round, also made of leather, and orange.

In all cases, the three balls can be thrown, caught or bounced, although with varying degrees of success.

A class generally has relationships with other classes.

In this video I will focus on how classes can extend and inherit properties and methods from other classes.

I will also explain how you can control inheritance between classes by defining access modifiers.

This class diagram shows four related classes.

The Ball class is the parent class that generically defines balls.

The Racquetball, Football and Basketball classes are subsets of the Ball class that define categories of specialized balls.

The Racquetball class is a blueprint for creating one or more racquetballs, which can be different colors and bounce at different speeds.

The Football class is a blueprint for creating American-style footballs, which are oval and can be different colors and materials.

Consider a pro, NFL football made of leather and a football made of the NERF material, which is commonly used in children's recreational balls to make them soft and spongy.

Class inheritance enables one class to extend another class.

In OOP terminology, the Racquetball, Football and Basketball classes are called the child or derived classes.

Since they are all sub-classes of the Ball class, they are said to extend the Ball class, or inherit from the Ball class.

The Ball class is being extended, or inherited from, and is often called the base, parent or super class.

In ActionScript 3, a class may only extend one other class.

Code that can be reused is commonly kept in the base class and then extended by other classes.

In the diagram, you can see that the Racquetball, Football and Basketball classes extend the Ball class to use its properties and methods.

For example, the Racquetball class has shape, material and color properties that it inherits from the Ball class.

By default, this class specializes in round and rubber balls, but does not define the color.

It also has properties of its own, like speed and weight that are not shared by the base Ball class.

Likewise, while the Racquetball class does inherit the throw(), catch() and bounce() methods, it also has its own method, hit().
When you create an instance of one of the child classes, it usually has access to both the properties and methods from the base class as well as the properties and methods defined within the child class itself.

You can, however, prevent instances of a child class from inheriting properties and methods by explicitly defining access modifiers on the class members.

In ActionScript 3, the access modifiers are internal, private, protected and public.

I will discuss these modifiers in more detail later in this video, but for now note that, by default, all properties and methods are set to internal, which means that child class instances do inherit the parent classes class members.

Properties and methods that are set to protected and public are also inherited while those set to private are not.

Sometimes you will want to inherit methods of a base class, but then modify them in your child class for that child class' specific needs.

This is called overriding the base class methods.

To override the behavior of a base class method, you create a new method within the child class with the same name.

The new method must have:

- The same name
- The same access modifier
- The same parameter list
- The same parameter data types
- The same return type
- and must be explicitly marked with the override attribute

Within the new function, you must also call the super() method which directs this child method to invoke the functionality of the super class for use in this method.

Note that the overriding method can do nearly everything the overridden method can do plus you can add your own code to do more.

You will learn more about these requirements in later videos.

For now focus on the concept that a child class can both inherit a method from a parent class and modify it for its specific needs.

Now, back to my ball example.

You can see that the bounce() methods in the Football and Basketball classes inherit directly from the bounce() method in the Ball class.

However, since an American-style football is oval and does not bounce easily, I can change the functionality of its bounce to be more similar to a spike, which occurs when a player hurls the ball at the ground, usually in celebration.

A basketball bounces much more easily, but I could modify its bounce() method to work with more specific rules that define a dribble.

Let me quickly review some of the OOP terminology and concepts that you have learned in this video.

The Ball class contains information and behaviors, known as properties and methods, which are inherited by the Racquetball, Football and Basketball classes.

Therefore, the Ball class is commonly referred to as the super, parent or base class.

The three classes that are derived from the Ball class are known as child or derived classes and are said to extend or inherit from the parent class.

By default, properties and methods from the parent class are inherited by the child class, but you can prevent inheritance by defining access modifiers which I will discuss more in this video series.

The child class can use the inherited methods as defined in the parent class or it can override the inherited method to create additional or replacement functionality.

Lastly, remember that the goal of creating these classes is to use them as templates for creating objects or instances that can interact in your application.

This image shows three instances of the Racquetball class.

Each racquetball instance has all the properties of the class and can use the methods defined in the class.

In Flex development, we often refer to the MXML code we write as “tags”.

For instance, we call the Application MXML code the “Application tag”.

This makes the code more approachable in some ways, but in reality the “Application tag” is actually an instance of the Application ActionScript class, made available in XML form.

You can create instances of any ActionScript class by using them in your application.

You can also extend a class by creating your own custom components, properties and method.

In Flash Builder, I am opening Help > Flash Builder Help to open the Adobe Community Help window.

Note that the first time you use the Community Help tool, you may be asked to synch your local content with the latest available content, which may take some time.

I am searching for the Spark List component and expanding the Search Options to constrain the search results to the ActionScript 3.0 Reference.

The ActionScript Language Reference material, by the way, is also known as AS Docs.

I am clicking on the first result that is returned to access the documentation on the Spark List component and then clicking on the dividing bar to maximize the viewing area.

Once you learn how to use AS Docs, you will likely find that it is an invaluable tool for your development needs.

If you remember that all MXML tags are actually classes, then you should now be familiar with at least some of the terminology used in AS Docs.

The inheritance section tells you from which super classes the List component derives.

You can clearly see that classes can fall into a long chain of parent/child relationships.

Here you can see some of the class properties that define each List component instance.

The links in the upper right of the page, are quick links to the sections of this page.

I am clicking on the Methods link to locate all of the methods available in this class.

You can see that there are some methods listed in the Protected Methods section and a link to expand the list to also show the inherited methods.

This was just a quick introduction to the AS Docs. I will show you more details about the different reference materials, like styles and skin parts as the video series progresses.

Now that you are familiar with the idea that MXML tags are actually classes, let me show you how this is implemented in your Flex code.

In this next section, I will create this Flex application that displays two employees.

I will create some instances of MXML classes and create my own custom MXML class with custom properties to display the image and name for each employee.

This is the starter file for the application I am about to build.

I am double-clicking on the Editor tab to maximize the window so that you can see more code.

The application only contains the Application tag set and the comments that identify where I should place my code based on the coding convention that I am following in this video series.

I am minimizing the Editor tab and switching to Design mode.

In the Components view, I am looking in the Controls branch to locate the Label control.

I am dragging an instance of the Label control and placing it in the upper left corner of the Design area.

Remember that, in OOP terms, the Label display in the Components view is really just a representation of the Label class.

The Label control that I placed in the Design area is one instance of that class.

I can modify the instance properties in the Flex Properties view.

I am setting a value of the text property to Employee Portal, and then changing the width property to 300, and the height property to 40.

I am returning to Source mode and, under the Styles comment, I am adding a Style instance and setting the source property to Styles.css.

This references the Styles.css file, which contains the custom selector, titleHeader, that you created in a previous exercise.

Back in the main application file, I am locating the Label control and adding the styleName property with a value of titleHeader.

When I save the file and return to Design mode, you can see that the style is reflected in the Design area.

From the Components view, I am dragging an Image control and dropping it into the Design area below the Employee Portal header.

In the Properties view, I am clicking on the folder icon to the right of the Source property to locate the images directory in this project.

I am selecting the source folder, then the images folder, then aparker.jpg.

Now I am clicking Open.

I am returning to Source mode and locating the Image control.

It is a best practice to only use the Image control if you need to skin an image by, say, adding a border to it. Since I just want to display an image in this exercise, I am changing the Image control to the lighter-weight BitmapImage control.

I am returning to Design mode to show you that the image is still properly displayed.
I want to use another instance of the Label control to display the employee's name, so back in Design mode, I am dragging and dropping another instance of the Label into Design view below the Image control.

In the Properties view, I am assigning this instance of the Label control a property value of Athena Parker.

I am clicking the Enter key so the value in the text property is accepted.

You may recall that at the beginning of this video I said that OOP models the real world, where classes are categories that define elements and objects are specific instances of these classes that are created from the class blueprint to work with other objects.

In this example, I have created two instances of the Label class.

The first one displays the text Employee Portal and the second the name of the employee.

I have also created an instance of the Image class that has a property that specifies the image to display.

When I save the file and run the application, Flash Builder builds and compiles the application and you can see that the application appears as it does in Design mode.

In the last few steps you watched me create this display that contains the Employee Portal header and Athena's name and picture.

Since there is more than one employee in this company, I know that I want to re-use the display of the employee’s image and name.

I am going to turn this display into my own custom MXML component, named EmployeeDisplay.mxml and then reuse it as many times as I like.

Of course, it’s not very useful for me to show the same image and name over and over again, so I will also show you how to create custom class properties for my new EmployeeDisplay class and pass in the appropriate values for the image and name so that each instance displays a unique person.
You learned earlier that a child class that is based on a parent class is said to extend the parent class.

The extended class benefits from all of the properties and methods of the parent class but can also add its own specific requirements.

You also learned about the Group class in a previous video so you know that it is a container, which holds other components for display.

I am going to use the Group class as the base of my new custom component.

By convention, many developers will store all of their custom component class files in a subfolder of the src package, which contains the main Application file.

In these videos, and in the related exercises, you will use a subfolder named components.

In Flash Builder, I am right-clicking on the src folder in the Package Explorer and then selecting New > Folder, which you cannot see since it is outside the recording area.

In the New Folder dialog, I am naming the folder components and then clicking Finish.

You can see the new folder inside of the src package.
Now I want to create the custom component MXML class file.

I am right-clicking on the components folder and selecting New > MXML Component.

In the New MXML Component dialog, I am naming the new component EmployeeDisplay.

You can see that the new component will be saved in the components package.

The Based on value is already set to the Group container in the Spark package.

I want the component to resize itself based on its content, so I will delete the hard-coded width and height values.

When I click Finish, you can see the new MXML file displayed in the Editor view.

Note that this file looks very similar to the default main application.

The main difference, of course, is that the main application file uses an instance of the Application class as the base of the application while this custom component uses the Group class as the base of this new class.

I am returning to the main application file to cut the BitmapImage and Label control instances that are specific to the employee.

I am pasting them in the EmployeeDisplay.mxml custom component file.

Now these elements are part of the custom component rather than part of the main application.

I'm going to go ahead and clean up this file a little bit, and the application file, to place everything in the appropriate locations based on the coding conventions for this course.

You use custom components similarly to Flex framework components.

You can see here that the EmployeeDisplay component is typed in MXML with x and y properties.

The only difference is that it does not use the MX or Spark namespace.

It references the components namespace, which is defined in the main application as pointing to the components.* package, or more literally, the components directory that I created earlier.

Remember that the asterisk in the Editor tab means that the file has not been saved.

I am saving the file and returning to the main application.

I am replacing the employee image and name that used to be in the main application with the new custom component instance.

In addition to the Flex framework classes, Flash Builder also shows you context-sensitive help for your custom components classes.

I am typing the EmployeeDisplay component.

As I do that, you can see that Flash Builder locates the component and recognizes that it is in the components folder.

When I hit the Return key to accept that selection, Flash Builder inserts the code as expected and also creates the namespace reference in the main Application tag.

I need to close my custom component tag instance and then I am saving the file and switching to Design mode, where you can see that the EmployeeDisplay instance is displayed in the Design area.

Notice, though, that the component outline is larger than the actual content.

When I return to the EmployeeDisplay.mxml file, you can see that it has x and y values defined for the controls.

I am selecting both controls and switching to Design mode for the custom component to reposition the elements using the arrow keys.

I am saving the file and returning to the main application file where you can see that I need to reposition the custom component.

From the Components view's Custom branch, I am dragging another instance of the EmployeeDisplay custom component to the Design area and lining it up with the first instance.

When I switch back to Source mode, you can see that the first component instance now has an x and y properties and there is a second instance.

The first one uses a single tag syntax, while the second one uses a tag block syntax.

When I save both files and run the application, you can see both instances of the custom component are shown.
Now, I need to update the custom component to display different information for two different employees.

To create and use a class property in an MXML custom component, you must:

First create a Script tag set in the MXML component file for your ActionScript code.

You will also need to define the class property as a variable with an access modifier, and of course, use the property as appropriate to the task.

Lastly, you must pass in a value for that property from the instantiation of the custom component.

I will start this process in the MXML custom component file by double-clicking on the Editor tab to maximize the window so you can see more code.

You create class properties in ActionScript.

So far in this video series, you have used MXML code but not very much ActionScript code.

Remember that the Flex framework contains both programming languages, where MXML is geared towards display elements and ActionScript is focused on application logic.

Often you will place both MXML code and ActionScript code in the same file.

When this happens, the ActionScript code, is placed inside of a Script tag block, which is part of the fx namespace.

I am creating the Script block towards the top of my MXML file.

Notice that Flash Builder automatically generates a CDATA block, which is an XML directive to treat the content inside the block as non-XML elements.

Inside the Script block, I am using the var keyword to declare that I am creating a variable.

Next, I am giving the variable a name. I want to pass in the name of the image to use for the employee's image, so I am naming the variable imageFile.

Since it will be a string value, I am using post-colon syntax to data type the variable to a String.

I end every line of ActionScript with a semicolon to denote that the command is over.

Remember from our discussion earlier, that all properties and methods of a class are, by default, inherited by their children unless explicitly prevented by the access modifier.

The access modifier also defines how a property or method may be accessed by other class instances.

By default, since I did not declare an access modifier for my variable, it will use the internal access modifier, which means it can be accessed only by instances of other classes that were created in the same package.

You will learn about packages in the next video.

For now, simply understand that this means the property can only be accessed by instances of other classes defined in the components directory.

A private modifier declares that the property can only be accessed by code in the same class.

A protected modifier declares that the property can only be accessed by code in the same class or any class which inherits from this class.

Lastly, the public modifier defines that the property can be accessed by any other code.

I want to be able to access this property from the main application, so I will set its access modifier to public.

It is considered a best practice to always explicitly define the access modifier for all of your class members.

The imageFile variable will be used to accept the name of the image file for the employee.

I also need a variable that will accept the name of the employee.

Under the imageFile variable, I am typing fullName and pressing CTRL+1 to invoke Flash Builder's quick assist tool and selecting the Create instance variable 'fullName' option. Like the content assist tool, the quick assist tool makes coding a little easier.

This creates a private variable data typed to the Object class, but I want it to be a public variable data typed to the String class. I am adding a semi-colon to the end.

I want to use these values in the BitmapImage and Label controls, respectively.

Therefore, I am replacing the hard-coded value for the image and the full name with binding directives.

When I save the custom component file, remember that Flash Builder will automatically attempt to compile the application to let me know if it encounters any issues.

You can see that two warnings appear that state that data binding will not detect the assignments to these variables.

You will need to use the Bindable metadata directive to bind an ActionScript variable to an MXML control.

In the ActionScript block, I am using content assist to add the [Bindable] metadata directive before each of the public variables.

I am saving the file so that Flash Builder recognizes my changes and recompiles the application to remove the warning messages.

The last step is to pass the correct values from the main application to each of the custom component instances.

I am returning to the main application file and locating the two instances of this component.

In the first instance, I’m moving the properties to the next line and then adding the imageFile property.

Flash Builder, as you can see, can locate this custom property in the code hinting options.

I am setting the value to aparker.jpg, which is the name of the image file with Athena's picture.

I am also adding the fullName property with a value of Athena Parker.

To the second instance, I am adding the imageFile property with a value of stucker.jpg and a fullName property with a value of Saul Tucker.

Now, I just wanted to show you that, in order to force the code assist tool to appear, you can hold down the Control key and press the Spacebar at the same time.

When I save the file and run the application, you can now see two instances of the EmployeeDisplay custom component, each displaying a different employee.

For your next step, work through the exercise titled “Creating MXML custom components with ActionScript properties”.