In this session, you will build a RIA using ColdFusion for server-side resources.
Introducing the session

What you will learn

- Understand the Adobe Flex 2 product family
- Create and run a simple Flex application that uses data bindings
- Layout a Flex application with containers
- Use components to properly architect a Flex application
- Handle events and create event handler functions
- Retrieve remote data from ColdFusion components
- Populate a ComboBox with remote data
- Use a wizard to build CFCs and an ActionScript class
- Retrieve value objects from CFCs
- Loop over an ArrayCollection, which is populated with remote data
- Filter data in an ArrayCollection
- Use custom events to dispatch data from custom components
- Use a view state to alter the display of data in the application
- Use a transition between view state

How you will learn it

- Slides
- Lecture
- Hand-on work
Understanding Flex

The following sections will help you understand why Flex is needed and what makes up the Flex product family.

Why Flex

Evolution of applications

Figure 1: The need for RIAs

The strength of the RIA
Flex product family

- The Flex product line provides the next generation of developer tools to build and deploy RIAs on the Flash Platform
  - Flex is an umbrella term for all the technologies of the Flex product line
- Product line consists of
  - Adobe Flex Builder 2
  - Adobe Flex 2 SDK
  - Adobe Flex Data Services 2
  - Adobe Flex Charting 2

Figure 2: Combining best features
Flex application framework

Figure 3: The pieces of the framework

Flex application process flow

Figure 4: From code to a SWF
Creating a simple Flex application with components

- Skeleton of an MXML application file contains
  - XML document declaration
  - Opening and closing <mx:Application> tags

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
  <!-- body of application here -->
</mx:Application>
```

- `<mx:Application>` is the default container tag

Specifying a namespace

- Use xmlns:mx attribute for the namespace, which defines the collection of legal MXML tags
- `http://www.adobe.com/2006/mxml` is a Uniform Resource Identifier (URI) which associates a prefix (in this case mx) with a manifest file in `flex-config.xml`

```xml
<?xml version="1.0" encoding="utf-8"?>
<namespace>
  <uri>http://www.adobe.com/2006/mxml</uri>
  <manifest>mxml-manifest.xml</manifest>
</namespace>
```

- `flex-config.xml` located in `installdirectory\Adobe\Flex Builder 2\Flex SDK 2\frameworks\flex-config.xml`
Manifest file

- Manifest file has a long list of tags, partially shown here

```xml
<?xml version="1.0"?>
<componentPackage>
  <component id="Accordion"
    class="mx.containers/Accordion"/>
  <component id="AnimateProperty"
    class="mx.effects/AnimateProperty"/>
  <component id="Application"
    class="mx.core/Application"/>
  <component id="ApplicationControlBar"
    class="mx.containers/ApplicationControlBar"/>
  <component id="ArrayCollection"
    class="mx.collections/ArrayCollection"/>
  
  <component id="mxml-manifest.xml"
    class="local/Adise\Flex Builder 2\Flex SDK 2\frameworks"
   />
</componentPackage>
```

- Manifest file, mxml-manifest.xml, is also located in

```
installdirectory\Adobe\Flex Builder 2\Flex SDK 2\frameworks
```

Setting the layout property

- The layout property defaults to your last selection when building an application or component from the appropriate dialog box
  - absolute: Visual elements must have specific x,y values or constraints for positioning
  - horizontal: Visual elements presented horizontally
  - vertical: Visual elements presented vertically

- If the layout property is not used in the Application tag, the default layout is vertical

Adding controls

- Used to display anything in an application
  - Text, text input, buttons, images, drop downs, etc.
  - Add between the application tags
  - Also referred to as components
  - Example of the Label control to display a single line of text

```
<mx:Label text="Hello Flexers!"/>
```

Specifying components’ properties

- Component properties can be specified in two ways
As attributes

```
<mx:Label text="Hello"/>
```

As nested tags or nodes

```
<mx:Label>
  <mx:text>Hello</mx:text>
</mx:Label>
```

## Displaying images

- Use the Image control to incorporate
  - JPEGs (non-progressive)
  - GIFs (can be transparent)
  - PNGs (can be transparent)
  - SWFs
  - SVGs (can only embed at compile time)

## Methods for displaying images

- Two ways to display images
  - Load them dynamically at runtime
  - Embed them in the SWF file
- Specify image source three ways
  - As source of an Image control; will load image dynamically at runtime
  - Use the Image `load()` method to switch the image dynamically at runtime
  - Embed them in the SWF at compile time using `@Embed`; useful for when you need instantaneous loading and offline applications

## Loading dynamically at runtime

- Specify the image using the `source` attribute of the `<mx:Image>` tag
  - Image is loaded at runtime

```
<mx:Image source="myimage.jpg"/>
```

- Assign an `id` to the image to reference in ActionScript

```
<mx:Image source="addis.jpg" id="imageID"/>
```

## Using the `load()` method

- Dynamically switch out the image using the `load()` method of the Image class
Code example

```xml
<?xml version="1.0"?>
<mx:Application backgroundColor="#FFFFFF"
xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Image source="addis.jpg"
    id="imageID" maintainAspectRatio="false"
    width="250" height="250"/>
  <mx:Button label="Change Pic"
    click="imageID.load('fugakyu.jpg')"/>
</mx:Application>
```

**Embedding images at compile time**

- You can embed images at compile time for instantaneous loading
- To embed, use the `@Embed` directive when specifying the image source

```xml
<mx:Image
  source="@Embed('../images/blue1.svg')"/>
```

**Creating a binding**

- Data binding is the process of binding the data of one object to another object
- Two ways to perform a binding
  - Curly braces (`{ }`)
  - `<mx:Binding>` tag

**Assigning instance names**

- Most MXML tags refer to classes
  - When you use the tag, you create an instance of the class
- Use the `id` property to assign an instance (or object) name
  - Allows you to refer to the object created

```xml
<mx:Label id="labelOne" text="Hello"/>
```

**Creating data bindings**

- Using curly braces
  - Assigns a property a dynamic instead of literal value
• Uses a broadcast/listener method

```xml
<mx:Label id="labelOne" text="hello"/>
<mx:Label text="{labelOne.text}"/>
```

• Using `<mx:Binding>` tag

```xml
<mx:Label id="labelTwo" text="hello"/>
<mx:Label id="labelThree"/>
<mx:Binding source="labelTwo.text" destination="labelThree.text"/>
```
Walkthrough 1: Creating a simple Flex Application

In this walkthrough, you will perform the following tasks:
- Create a Flex Builder project
- Create a simple Flex application
- Use a binding with a variable

Steps

Create a Flex Builder project

1. From the Flex Builder menu, select File > New > Flex Project.
2. Select ColdFusion Flash Remoting Service.
3. Click Next.
4. Enter the root folder and root URL of the ColdFusion server. In the default course installation, the settings are:
   - Root folder: C:\ad003h\servers\cfusion\cfusion-ear\cfusion-war\n   - Root URL: http://localhost:8300/
5. Click Next.

Figure 5: Specifying the ColdFusion server location
6. Set the **Project name** to be `AD003H` and the **Folder** to be `C:\ad003h\lab`.

7. Click Next.

8. Specify the Main application file to be `FStop.mxml`.

9. For the Output folder, specify `lab\bin`.

---

**Figure 6: Specifying project details**

**Figure 7: Specifying the main application and output**
10. Click Finish to create the project.

11. Set the Application's backgroundColor to black.

    backgroundColor="#000000"

**Use a binding**


13. In the Script block, create a bindable, private variable named `categories` datatyped as `String` and set it equal to `Test value`.

    <mx:Script>
    <![CDATA[
        [Bindable]
        private var categories:String="Test Value";
    ]]>  
    </mx:Script>

14. Below the Script block, insert an `<mx:Label>` tag and bind the `text` property to the `categories` variable. Set the `fontSize` to 20 and the `color` to `#FFFFFF`.

    `<mx:Label text="{categories}"  
              fontSize="20"  
              color="#FFFFFF"/>

15. Run the application.
    You should see the text displayed.

**Display an image**

16. Below the Label tag, insert an `<mx:Image>` tag with the `source` property set equal to `assets/header.jpg`. Set the location of the image at the x,y position 137,25.

    `<mx:Image source="assets/header.jpg"  
              x="137"  
              y="25"/>

17. Run the application.
    You should see the image displayed, along with the text.

18. Remove the Label tag and save the document.
Understanding Containers

- There are two general types of components:
  - Controls - Form elements, such as buttons, text fields, and list boxes.
  - Containers - Rectangular regions of the screen that contain controls and other containers.
- There are two types of containers:
  - Layout containers - Control the sizing and positioning of the child elements within them.
  - Navigator containers - Control user movement, or navigation, among multiple child containers.
- This session discusses only layout containers.

Using Layout containers

- A component which can hold other components
- A Flex tool for laying out visual elements into logical groupings

Figure 8: The final container layout for the Cafe Townsend application
Figure 9: The final Cafe Townsend application
Architecting an application with MXML components

- In general, you don’t want all your application code in a single file
- Best practice to break your application into separate and multiple MXML files
- Breaking up your application has the following benefits:
  - Allows multiple developers to work on the application
  - Makes it easier to maintain the application
  - Application is better and more logically structured into smaller pieces
  - Enables you to reuse parts of the application elsewhere

Using MXML components

- When you separate your application into multiple MXML files:
  - Each file is an “MXML component”
  - The main MXML application loads the individual components
  - Each component often represents a different state of the application
  - There is one Application tag in the main MXML doc; everything else is a component

![Diagram](image)

Each logical block of code for each tab would be a different component.

**Figure 10: How components could be used with a TabNavigator**

Creating MXML components

To create separate MXML components:
- Create a new file
Instead of using an `<mx:Application>` tag, use any another component tag such as VBox, Form, Button, etc.

Avoid using any unnecessary containers for performance reasons

In the following component, it would be better just to use the Canvas as the base tag, instead of a Canvas inside a VBox

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:VBox
xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Canvas>
  </mx:Canvas>
</mx:VBox>
```

Instantiating MXML components

Once you have broken your application into separate components, you want to instantiate those components within the MXML application

Add an XML namespace

Use `*` to represent all the components in a particular folder

For example `xmlns[:prefix]="subfoldername.*"`

```xml
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:comp="components.*"/>
```

Instantiate component like any other MXML tag

Use correct namespace

```xml
<comp:Component1/>
```
Demonstration 1: Using properties and methods of components

In this demonstration, your instructor will perform the following tasks:

- Use a component's property and method.

Steps

1. Open Demo_Components.mxml.
2. Run the application.
   Notice use of property and method of the component.
Walkthrough 2: Creating components

In this walkthrough, you will perform the following tasks:

- Create two components
- Create a property of a component
- Instantiate a component

Steps

1. Right click on the AD003H project and select New > Folder.
2. Set the Folder Name to be views, then click Finish.

Create a component and create a property of the component

3. Right click on the views folder then select New > MXML Component.
4. Set the Filename of the component to be SelectPhoto.mxml. Base the component on a VBox. Remove any Width and Height settings. Click Finish.
5. Below the opening <mx:VBox> tag insert an <mx:Script> block.
6. In the Script block, create a bindable, public variable named categories datatyped as String.
7. Below the Script block, insert an <mx:Label> tag and bind the text property to the categories variable.

Instantiate the new component

8. Open FStop.mxml.
9. In the Application tag, add an XML namespace that permits you to access the components in the views folder using the letter v as the prefix.

    xmlns:v="views.*"

10. Just above the end <mx:Application> tag, instantiate the SelectPhoto component. Give the component the instance name selectPhoto.

    <v:SelectPhoto id="selectPhoto"/>

11. To the categories property, bind the categories variable.

    Note: Use tag help to get the categories property inserted.
12. Set the location of the component at the x,y position 137,112.
13. Run the application.
   You should see the text displayed below the header graphic.

**Create another component**

14. Right click on the *views* folder then select **New > MXML Component**.
15. Set the **Filename** of the component to be *DisplayPhoto.mxml*. Base the component on a *VBox*. Remove any **Width** and **Height** settings. Click **Finish**.
17. Return to *FStop.mxml*.
18. Just above the end `<mx:Application>` tag, instantiate the **DisplayPhoto** component. Give the component the instance name `displayPhoto`.

   ```xml
   <v:DisplayPhoto id="displayPhoto"/>
   ```

19. Set the location of the component at the x,y position 240,147.

*Note: At this point this is all you will do with this component.*

20. Be sure all files are saved.
Understanding events

- Events let the developer know when something happens within a Flex application
- System
  - Occur as result of systematic code execution
  - e.g., creationComplete, initialize, show
- User
  - Occur as a result of user interaction with a control or container in the application
  - e.g., click, change, mouseOver

System events

- Occur through the execution of code
- The following occur for every object that is a subclass of UIObject

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creationComplete</td>
<td>Broadcast when an object has finished its construction, property processing, measuring, layout and drawing</td>
</tr>
<tr>
<td></td>
<td>Set default startup settings on this event, you know all objects are created and ready to use</td>
</tr>
<tr>
<td></td>
<td>Commonly used with the Application object</td>
</tr>
<tr>
<td>initialize</td>
<td>Broadcasts when the component has finished its construction and has all initialization properties set</td>
</tr>
<tr>
<td></td>
<td>The initialize event can occur too soon to interact properly with child objects. Care must be taken not to execute statements on undefined objects with the initialize event.</td>
</tr>
<tr>
<td>result</td>
<td>The result event is dispatched when a service call successfully returns data</td>
</tr>
</tbody>
</table>

*Note: Events on the Application object fire last*

User events

- Supports subset of DOM Level 3 events
Table 2: Some user events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>click</td>
<td>A pointing device button is clicked over an element</td>
</tr>
<tr>
<td>change</td>
<td>An event that occurs when the value of the control changes, such as each key typed into a TextInput control or a new selection made in a ComboBox control</td>
</tr>
<tr>
<td>mouseDown</td>
<td>A pointing device button is pressed over an element</td>
</tr>
</tbody>
</table>
Demonstration 2: Viewing system and user events

In this demonstration, your instructor will perform the following tasks:

- Display system event
- Display user events

Steps

Note: At this point just view what the application does and not look at the code. Later in the module the code will be examined.

1. Open Demo_Events.mxml.
2. Run the application.
   Notice the system events that occur without user interaction.
3. Interact with the button labeled Test to see user events.
Creating event handlers using inline ActionScript

- Event handlers, also called event listeners, are the code you write to respond to events
- Up to now, the values of attributes and properties have been scalar values, except for bindings

```xml
<mx:Button label="Click Me"/>
```

- When using an event, the value associated with it is ActionScript, which is called an event handler

```xml
<mx:TagName eventName="ActionScript code"/>
```

- Two ways to specify an event handler
  - Using inline ActionScript
  - Using ActionScript functions in a script block

Creating event handlers using inline ActionScript

- Write ActionScript inside a tag

```xml
<mx:TagName eventName="[ActionScript statement or function call here]"/>
```

- Example: Responding to a Button click event

```xml
<mx:Button label="Click Me"
    click="myLabel.text='You clicked'"/>
```

```xml
<mx:Button id="myButton" label="Click Me"
    click="myLabel.text='You clicked'"/>
```

Using ActionScript in MXML tags

- ActionScript can only be used as a value for an event, not properties
  - Bindings ({{}}) permit you to use ActionScript as property values

```xml
<mx:Button id="myButton" label="Click Me"
    click="myLabel.text='You clicked'"/>
```

```xml
<mx:Label id="myLabel" text="Click {myButton.label}"/>
```
Handling events with ActionScript functions

- If you need more than one line of code to handle an event, place the code within a function
- Enables reuse
- Keeps code cleaner
- Place code with an `<mx:Script>` tag
- Everything inside Script block treated as XML by parser
- Wrap code in CDATA to keep special characters from being processed

Defining a function in an MXML document

```
<mx:Script>
  <![CDATA[
    private function fillLabelControl():void
    {
      myLabel.text="You Clicked";
    }
  ]]> 
</mx:Script>

<mx:Button label="Click Me"
  click="fillLabelControl()"/>
<mx:Label id="myLabel" />
```
Understanding the event object

- Every event object is an instance of the flash.events.Event class
- Each time an event occurs, an event object is created
- Contains a large collection of named properties
  - Some are standard properties, others depend on the event that occurs
  - Refer to this object inside an event handler as an object named `event`

```xml
<mx:Button label="Click Me" id="button1"
    click="fillLabelControl(event)"/>
```

Event object properties

- Every event object contains properties
  - Some are the same for all events (type and target)
  - Some are specific to the event broadcast
  - Some are custom properties the developer specifies

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>String</td>
<td>The name of the event for example, click</td>
</tr>
<tr>
<td>target</td>
<td>Event</td>
<td>The component instance that broadcast the event</td>
</tr>
<tr>
<td>target.id</td>
<td>String</td>
<td>A property of the target property object, which is the instance name of the target</td>
</tr>
</tbody>
</table>

Data typing the event object

- Use the Event class as the datatype when needed

```typescript
private function aHandler(event:Event):void
```

- Stricter datatypes are available that are subclasses of Event
  - MouseEvent
  - DragDropEvent
  - ResultEvent
  - Custom events (discussed later in this course)
- Benefits of datatyping as strictly as possible
  - Faster runtime performance
  - Compile time type checking
  - Access to event specific properties
- If stricter datatyping NOT used, may not have access to properties specific to the specific event
All event classes, except the basic Event class, need to be imported to be used.

**currentTarget versus target**

- There are times when it is more convenient to handle an event not on the actual object (or target) that dispatched the event, but on an object that contains the target.
- In Flex’s event propagation, you can choose to handle the event at the target or in a container of the target.
- Referred to as event bubbling.
- Every Event object has a `target` and a `currentTarget` property that help you to keep track of where it is in the process of propagation.
  - The `target` property refers to the dispatcher of the event.
  - The `currentTarget` property refers to the current object that is being examined for the event.
- If listening for an event on a container of the target, use the `currentTarget` property to access the event object’s details.

![Figure 11: When to use currentTarget and target properties](image.png)
Walkthrough 3: Handling an event and examining the event object

In this walkthrough, you will perform the following tasks:
- Add a Button object with a click event
- Handle the click event
- Examine the event object

Steps

1. Open SelectPhoto.mxml.
2. Remove the Label in the component.

Add a Button with a click event

3. Below the Script block, add an `<mx:Button>` tag. Set the `label` property equal to Click Here.
4. Have the `click` event call a function named `dispatchPhoto()`. Pass the event object as a parameter.
   ```xml
   <mx:Button label="Click Here"
               click="dispatchPhoto(event)"/>
   ```

Handle the event

5. In the Script block below the variable declaration, insert a private function name `dispatchPhoto()`. The function should accept a parameter named `event` datatyped as `MouseEvent`.

   ```actionscript
   private function dispatchPhoto(event:MouseEvent):void
   {
   }
   ```

Debug the application

6. Place a breakpoint on the closing brace of the function.
7. Debug the application and click the Button to call the event handler.
8. Return to Flex Builder 2 in the Flex Debugging perspective and double click the Variables view to enlarge it.
9. View the `event` object and note the `target` and `currentTarget`.
10. Terminate the debugging session.
11. Remove the `<mx:Button>` tag from the component as it will no longer be needed.
12. Return to the Flex Development perspective.
Retrieving Remote Data

- Data retrieval overview
  - Retrieve data via HTTP
    - Use `<mx:HTTPService>`
    - Very often XML data
  - Consume web services
    - Use `<mx:WebService>`
  - Call remote methods from Java objects or CFCs
    - Use `<mx:RemoteObject>`
      - Best performing option using ActionMessageFormat (binary protocol)
- Use the `<mx:RemoteObject>` tag to connect to remote objects, like ColdFusion components

```xml
<mx:RemoteObject
  id = "id"
  destination = "ColdFusion"
  source = "path to CFC"
  result = "resultHandler"
  fault = "faultHandler"/>
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Instance name of the object</td>
</tr>
<tr>
<td>destination</td>
<td>Destination of the Service; should match an entry in the services-config.xml file</td>
</tr>
<tr>
<td>source</td>
<td>Dot-separated path from the webroot (or contextroot) of the CF application to the component. Used only with unnamed services (see below)</td>
</tr>
<tr>
<td>result</td>
<td>result handler ActionScript code</td>
</tr>
<tr>
<td>fault</td>
<td>fault handler ActionScript code</td>
</tr>
</tbody>
</table>

Invoking methods of a remote object service

- Call public methods of the remote object upon Flex application events
  - User Event - Button click event
System Event - The `<mx:Application>` tag’s `creationComplete` event

`creationComplete="myService.getData()"`

Invoke methods as 'regular' AS object methods

```
<mx:RemoteObject
    id="myService"
    destination = "ColdFusion"
    source = "components.myCFC" />
```

```
<mx:Button
    label="get Data"
    click = "myService.getData()" />
```

### Using results of a remote object service

- The data that is returned as a result of a remote method call is placed in the `lastResult` property of the result object

```
myService.getData.lastResult
```

### The result event

- A remote object service method has a result event that is an instance of `mx.rpc.events.ResultEvent`; an event handler can be assigned to process that event

```
<mx:RemoteObject
    id="myService"
    destination = "ColdFusion"
    source = "components.myCFC"
    result = "myEventHandler(event)" />
```

- An event object (default or custom) is created for every event
- If the CFC returns data, it is in the `result` property of the event object
- The `result` property is cast as a generic ActionScript object and should be recast to its native type as needed

```
event.result as ArrayCollection
```

### Using query results returned from a CFC

- A query is automatically translated into an ActionScript `ArrayCollection`
Understanding the ArrayCollection results

- After a RemoteObject operation executes and a query is returned from the CFC, the returned data is placed in the operation's `lastResult` object
  - Represented as an ActionScript ArrayCollection
  - Flex interprets the data to appropriately represent base types
  - An ArrayCollection is a complex data structure that is returned
  - ArrayCollection recommended class to use as a `dataProvider`
  - Must import the class to use it

```actionscript
mx.collections:ArrayCollection
```

- Using an `ArrayCollection` class has the following advantages
  - The elements of the ArrayCollection can be used in bindings that will continue to be monitored
    - This is not the case with the normal `Array` class, once an element from an array is used in a binding it is no longer monitored
  - Implements both the ICollectionView and IList interfaces
  - Provides rich set of tools for data manipulation
  - Example to retrieve an item of the collection, where `n` is the zero-indexed position of the item

```actionscript
arrayCollectionInstance.getItemAt(n)
```
Walkthrough 4: Retrieving and using query data from a CFC

In this walkthrough, you will perform the following tasks:

▸ Create a RemoteObject object
▸ Call a remote method of a CFC
▸ Handle the result event and place the returned results into an ArrayCollection
▸ Pass the returned results to a component and display them using a DataGrid

Steps

1. Open the file C:\ad003hCF\servers\cfusion-ear\cfusion-war\WEB-INF\flex\services-config.xml.
2. Note the destination with an id of ColdFusion.
3. Open FStop.mxml.

Create a RemoteObject object

4. Just below the Script block, insert an <mx:RemoteObject> tag. Set the id to be catRO, the destination to be ColdFusion and the source to be cfcs.CategoryService.
5. Add a result event and have it call an event handler named getCategoriesHandler(), and pass the event object as a parameter.

```
<mx:RemoteObject id="catRO"
    destination="ColdFusion"
    source="cfcs.CategoryService"
    result="getCategoriesHandler(event)"/>
```

Call a remote method of the CFC

6. From the CFCs project, open the CFC CategoryService.cfc. Note that the getCategories function returns a query.
7. Return to FStop.mxml.
8. To the <mx:Application> tag add a creationComplete event that calls the getCategories() remote method is called from the catRO RemoteObject.

```
creationComplete="catRO.getCategories()"
```
Create a variable to hold the returned results

9. In the Script block, alter the `categories` variable declaration. Change it so the variable is datatyped as an ` ArrayCollection` and no longer assigned a value.

   *Note: When you datatype the variable Flex Builder should automatically import the ArrayCollection class.*

   ```
   import mx.collections.ArrayCollection;
   
   [Bindable]
   private var categories:ArrayCollection;
   ```

Build the event handler function

10. At the bottom of the Script block, insert an private function named `getCategoriesHandler()` datatyped as `void`. The function should accept a parameter named `event` datatyped as `ResultEvent`.

   *Note: When you datatype the parameter Flex Builder should automatically import the ResultEvent class.*

   ```
   private function getCategoriesHandler(event:ResultEvent):void
   {
   
   }
   ```

11. Place a breakpoint on the closing brace of the function and debug the application.
12. Be sure the result contains the ArrayCollection of category information.

![Figure 12: The results from the remote method call](image)

13. Terminate the debugging session and return to the Flex Development perspective.

14. In the function, assign the `categories` variable the values from `event.result`. You must also cast the results as an `ArrayCollection` using the `as` operator.

   
   ```
   categories = event.result as ArrayCollection;
   ```

**Use the results in the component**

15. Open SelectPhoto.mxml.

16. In the Script block, change the datatype of the `categories` variable to an `ArrayCollection`.

*Note: When you datatype the variable Flex Builder should automatically import the ArrayCollection class.*
17. Just below the Script block, insert an `<mx:DataGrid>` tag and bind the `dataProvider` property to the `categories` ArrayCollection.

```
<mx:DataGrid dataProvider="{categories}"/>
```

*Note: The DataGrid is a very useful and complex class. In this case you are just using it to dump out data to verify it has been passed to the component correctly.*

18. Run the application.
   
   You should see the category data displayed in a DataGrid.
19. Remove the DataGrid from the component.
20. Save all files.
Using the ComboBox control

- The ComboBox control contains a drop-down list from which the user can select a single value
  - Its functionality is very similar to that of the SELECT form element in HTML
- Supply data to list-based controls, like the ComboBox, using the dataProvider property
  - Datatype of the dataProvider often and ArrayCollection of objects
- Same property to supply data to all list-based controls
  - ComboBox
  - List
  - DataGrid
  - Tree
  - TileList
  - HorizontalList
- If the dataProvider is an object need to specify property to display using the labelField property

```xml
<mx:ComboBox id="category"
   dataProvider="{categories}
    labelField="catName"/>
```

- Can also use a labelFunction to build a more complex string to display
Walkthrough 5: Populating a ComboBox

In this walkthrough, you will perform the following tasks:

► Populate a ComboBox with data retrieved from a method of a CFC

Steps

1. Open SelectPhoto.mxml.

Populate the ComboBox with an ArrayCollection

2. Just below the Script block, insert an `<mx:ComboBox>` tag. Set the `id` property to `catSelect`, bind the `dataProvider` to the `categories` variable and set the `labelField` property to display the CATEGORY field.

   ```xml
   <mx:ComboBox id="catSelect"
               dataProvider="{categories}"
               labelField="CATEGORY"/>
   ```

3. Run the application.
   You should see the populated ComboBox and be able to select data from it.
Returning value objects from a CFC

- Value objects (also known as data transfer objects) are classes created free of implementation detail and business logic
- Contain only value of data related to an object
- Often used for transfer of data to a backend
- Formally defined in the Core J2EE Pattern catalogue
- Implemented as CFCs in ColdFusion
- Implemented as ActionScript classes in Flex

Map ActionScript class to CFC

- By mapping the ActionScript class to a CFC you can return value objects, rather than generic objects, from the CFC

ActionScript class

```actionscript
package valueObjects
{
    [RemoteClass(alias="Photo")]
    [Bindable]
    public class Photo
    {
        public var PHOTOID:Number = 0;
        public var FILENAME:String = "";
        public var PHOTOGRAPHER:String = "";
        public var DESC:String = "";
        public var CATEGORYID:Number = 0;

        public function Photo()
        {
        }
    }
}
```
CFC

▶ Partial example CFC value object

```coldfusion
<cfcomponent output="false" alias="Photo">
  <!---
These are properties that are exposed by
this CFC object.
These property definitions are used when
calling this CFC as a web services,
passed back to a flash movie, or when
generating documentation

NOTE: these cfproperty tags do not set any
default property values.
--->
  <cfproperty name="PHOTOID" type="numeric"
default="0">
  <cfproperty name="FILENAME" type="string"
default="">
  <cfproperty name="PHOTOGRAPHER" type="string"
default="">
  <cfproperty name="DESC" type="string"
default="">
  <cfproperty name="CATEGORYID" type="numeric"
default="0">
  </cfscript>

  <cfscript>
  //Initialize the CFC with the default
  properties values.
  variables.PHOTOID = 0;
  variables.FILENAME = "";
  variables.PHOTOGRAPHER = "";
  variables.DESC = "";
  variables.CATEGORYID = 0;
  </cfscript>

  <cffunction name="init" output="false"
  returntype="Photo">
    <cfreturn this>
  </cffunction>

  <cffunction name="getPHOTOID" output="false"
  access="public" returntype="any">
    <cfreturn variables.PHOTOID>
  </cffunction>

  <cffunction name="setPHOTOID" output="false"
  access="public" returntype="void">
```


<cfargument name="val" required="true">
<cfif (IsNumeric(arguments.val)) OR (arguments.val EQ "")>
  <cfset variables.PHOTOID = arguments.val>
<cfelse>
  <cfthrow message="'#{arguments.val}# is not a valid numeric"/>
</cfif>
</cffunction>
</cffunction>

<cffunction name="getFILENAME" output="false" access="public" returntype="any">
<cfreturn variables.FILENAME>
</cffunction>
</cffunction>

<cffunction name="setFILENAME" output="false" access="public" returntype="void">
<cfargument name="val" required="true">
<cfset variables.FILENAME = arguments.val>
</cffunction>
</cffunction>

Creating the value objects

- Wizards available to create both ActionScript classes and the CFCs
Walkthrough 6: Returning Photo value objects from a CFC

In this walkthrough, you will perform the following tasks:
- Use a wizard to create the value object ActionScript class and CFCs
- Create a RemoteObject to retrieve the photo value objects
- Handle the results by placing the returned value objects in an ArrayCollection

Steps

1. Right click on the AD003H project and create a new project named valueObjects.

Create the CFCs and ActionScript class

2. Open the RDS Dataview view.
3. Open the ad003hPhotos database, then the Tables folder.
4. Right click on the PUBLIC.PHOTOS table, then choose ColdFusion Wizards > Create CFC.

Figure 13: Starting the wizard

5. Set the CFC Folder by browsing to the CFCs project.
6. For the CFC Type select Bean CFC & DAO CFC.
7. Alter the Bean File Name, DAO File Name and Gateway File Name so PHOTOS is replaced by Photo in all three cases.
8. Check the box to create an ActionScript value object.
9. For the ASFolder browse to /AD003H/valueObjects.
10. Be sure the AS Package Name is valueObjects.
11. Check to be sure you dialog box appears as shown here.
12. Click Finish.

Create a RemoteObject for the PhotoGateway

13. Open FStop.mxml.

14. Just below the existing `<mx:RemoteObject>`, and another. Set the id to be `photoRO`, the destination to be ColdFusion and the source to be `cfcs.PhotoGateway`.

15. Add a result event and have it call an event handler named `getAllHandler()`, and pass the event object as a parameter.

```xml
<mx:RemoteObject id="photoRO"
    destination="ColdFusion"
    source="cfcs.PhotoGateway"
    result="getAllHandler(event)"/>
```
16. Add to the creationComplete event in the <mx:Application> tag so that the getAll() remote method is called from the photoRO RemoteObject.

   creationComplete="catRO.getCategories();
   photoRO.getAll()"

Create a variable to hold the returned results

17. In the Script block below the existing variable declaration, create a bindable, private variable named photos datatyped as an ArrayCollection.

   [Bindable]
   private var photos:ArrayCollection;

Build the event handler function

18. In the Script block below any existing functions, insert a private function named getAllHandler() datatyped as void. The function should accept a parameter named event datatyped as ResultEvent.

19. Place a breakpoint on the closing brace of the function and debug the application.

20. Note that the event.result is an Array of generic Objects, not an Array of Photo objects.

   Figure 15: Generic objects, not Photo objects returned

21. Terminate the debugging session.

Be sure the Photo class is used

22. In the Script block below the existing variable declaration, create a private variable named dummyPhoto datatyped as a Photo.
Note: When you datatype the variable Flex Builder should automatically import the Photo class.

```actionscript
import valueObjects.Photo;
private var dummyPhoto:Photo;
```

23. Debug the application again.
24. Note that the `event.result` is now an Array of Photo objects.

![Figure 16: An array of Photo objects returned](image)

25. Terminate the debugging session.

**Place the returned Array into an ArrayCollection**

26. In the `getAllHandler()` function, create a variable local to the function named `tempArray` datatyped as `Array`. Set this equal to `event.result` cast as an `Array`.

```actionscript
var tempArray:Array=event.result as Array;
```

27. Still in the function, set `photos` equal to a new ArrayCollection and pass `tempArray` as an argument to the ArrayCollection's constructor.

```actionscript
photos=new ArrayCollection(tempArray);
```

28. Use the debugger to be sure `photos` is populated correctly.
Looping through data using a Repeater

- Dynamically repeat MXML content at runtime
- Handle repetition of a small number of user-interface elements
- Repetition controlled by the `dataProvider` property, which will generally be an array of objects

Repeater component functionality

- Repeat controls or container
- Repeater component does not control the layout of repeated components
  - Usually place the Repeater inside a container
- Rules for using a Repeater
  - Use anywhere a control or container tag is allowed
  - Exception of repeating the `<mx:Application>` container
  - Can use more than one Repeater in an MXML document
  - Can nest Repeaters
  - Use only for objects that extend the `UIComponent` class
- Example

```mxml
<mx:Script>
<![CDATA[
   import mx.collections:ArrayCollection;
   [Bindable]
   private var myAra:ArrayCollection=new ArrayCollection(["halibut","redfish","seabass","tilapia","flounder"]);
]]>
</mx:Script>

<mx:Tile>
   <mx:Repeater dataProvider="{myAra}">
      <mx:Label text="In the repeater"/>
   </mx:Repeater>
</mx:Tile>
```

Figure 17: The output generated from the Repeater code
If the dataProvider is a subtag of the Repeater component only the code between the end of the dataProvider and the end of the Repeater is repeated.

```xml
<mx:Tile>
<mx:Repeater>
<mx:dataProvider>
<mx:String>halibut</mx:String>
<mx:String>redfish</mx:String>
<mx:String>talapia</mx:String>
<mx:String>flounder</mx:String>
</mx:dataProvider>

...<mx:Label text="In the repeater"/>
...
</mx:Repeater>
</mx:Tile>
```

*Figure 18: Repeated code when the dataProvider is a subtag*

- Can specify the `startIndex` and `count` properties to control the looping process.

### Using dataProvider data in repeated controls

- The main purpose of Repeater component is to create multiple components in the loop.
- Will often want to use data from the dataProvider so each component created in the loop is unique.
- For example a set of RadioButtons or CheckBoxes.

### Key Repeater class properties

- Two properties assist in accomplishing the task by accessing information from the Repeater and dataProvider during looping.
  - `currentIndex`: Number that specifies the element of the dataProvider being processed.
    - Zero indexed.
    - Undefined when looping complete.
  - `currentItem`: Reference to the object of the dataProvider being processed.
    - Undefined when looping is complete.
Example

```xml
<mx:Script>
<![CDATA[
    import mx.collections:ArrayCollection;
    [Bindable]
    private var myAra:ArrayCollection=new ArrayCollection(['halibut','redfish','sea bass','tilapia','flounder']);
]]>
</mx:Script>

<mx:Repeater id="myRepeater" dataProvider="{myAra}"
    <mx:Label text="{myRepeater.currentIndex}"/>
    <mx:Label text="{myRepeater.currentItem}"/>
</mx:Repeater>

0
halibut
1
redfish
2
sea bass
3
tilapia
4
flounder

Figure 19: Output from the code

Using currentItem with a complex data structure

- If the currentItem is an object and not a scalar value use object.property notation with currentItem
Example (The braces in the following code are not for bindings, but shorthand notation for object creation.)

```xml
<mx:Script>
<![CDATA[
import mx.collections.ArrayCollection;
private var arrayData:Array=
{name:"banana",cat:"fruit",cost:.99},
{name:"bread",cat:"bakery",cost:1.99},
{name:"orange",cat:"fruit",cost:.52},
{name:"donut",cat:"bakery",cost:.33},
{name:"apple",cat:"fruit",cost:1.05}];

private var dp:ArrayCollection=new ArrayCollection(arrayData);
]]>
</mx:Script>

<mx:Repeater id="myRepeater" dataProvider="{dp}"/>
<mx:Label text="{myRepeater.currentIndex}"/>
<mx:Label
text="{myRepeater.currentItem.name}: {myRepeater.currentItem.cost}"/>
</mx:Repeater>

</mx:Application>

0
banana: 0.99
1
bread: 1.99
2
orange: 0.52
3
donut: 0.33
4
apple: 1.05

Figure 20: Output from the code

Using Repeater data in ActionScript

After using the Repeater to create multiple components, you will often need to access data used in the Repeater to build the component
• Usually in an event handler
• On each iteration of the Repeater an event object is created containing data about that iteration and the data it used
• To access, use the `getRepeaterItem()` method to access the current item
• It returns the item in the dataProvider object that was used to create the component
• It is a method of the Repeater’s event object’s `target` property

```
event.target.getRepeaterItem()
```

• Is a method of UIComponent
Example

<mx:Script>
<![CDATA[
import mx.collections:ArrayCollection;
private var arrayData:Array=[
    {name:"banana",cat:"fruit",cost:.99},
    {name:"bread",cat:"bakery",cost:1.99},
    {name:"orange",cat:"fruit",cost:.52},
    {name:"donut",cat:"bakery",cost:.33},
    {name:"apple",cat:"fruit",cost:1.05}];

private var dp:ArrayCollection=new ArrayCollection(arrayData);

private function displayData(dataObj:Object):void {
    lab1.text=dataObj.name;
    lab2.text=dataObj.cost;
}
]]>
</mx:Script>

<mx:Label id="lab1" text="data here soon"/>
<mx:Label id="lab2" text="data here soon"/>

<mx:Repeater id="myRepeater"
    dataProvider="{dp}"
>
    <mx:Button
        label="{myRepeater.currentItem.name}: {myRepeater.currentItem.cost}"
        click="displayData(event.target.getRepeaterItem())"/>
</mx:Repeater>
Can also pass the event object to the function and use `getRepeaterItem()` method there

```actionscript
private function displayData(event:MouseEvent):void
{
    var dataObj:Object=new Object();
    dataObj=event.target.getRepeaterItem();
    lab1.text=dataObj.name;
    lab2.text=dataObj.cost;
}
```

**Understanding the differences between getRepeaterItem() and currentItem**

- Although `{myRepeater.currentItem}` binding and `getRepeaterItem()` method handle the same data, uses are different
- Use former in MXML data bindings and latter in ActionScript
- Cannot use binding expressions in event handlers - the following is invalid syntax
  ```actionscript
  mouseOver="getItem({myRepeater.currentItem})"
  ```
- Remember that `currentItem` and `currentIndex` only have values during the actual looping process; after the looping process the values are meaningless
Walkthrough 7: Displaying thumbnail images in a Repeater

In this walkthrough, you will perform the following tasks:
► Create a Tile container that contains a Repeater object
► Use the Repeater object to display thumbnail images

Steps

1. Open SelectPhoto.mxml.

Create a property to accept photo data from the main application

2. In the Script block below the other variable declarations, create a bindable, public variable named photos datatyped as ArrayCollection.

Display images from a Repeater nested in a Tile container

3. Below the ComboBox, insert an <mx:Tile> block.
4. Add a height property to the Tile and set it equal to 430 and a width property set equal to 750.
5. Add an <mx:Repeater> block nested in the Tile. Set the id equal to photoRepeater and bind the dataProvider bound to photos.
6. In the Repeater, insert an <mx:Image> tag. Set the source to assets/{photoRepeater.currentItem.FILENAME}, and the width and height to 80.

<mx:Tile height="430" width="750">
  <mx:Repeater id="photoRepeater"
    dataProvider="{photos}">
    <mx:Image source="assets/{photoRepeater.currentItem.FILENAME}"
      width="80" height="80"/>
  </mx:Repeater>
</mx:Tile>

Pass data to the SelectPhoto component

7. Open FStop.mxml.
8. Locate the instantiation of the SelectPhoto component (use the Outline view for help).
9. Add the `photos` property to the SelectPhoto component and bind it to the `photos` ArrayCollection.

10. Run the application.

You should see thumbnails of images displayed in the Tile.
Filtering data in an ArrayCollection

- Create a filter function that limits the ICollectionView to a subset
- Must also apply the function by assigning it to the filterFunction property of an ArrayCollection

Creating the function

- Function takes a single object as a parameter
  - Corresponds to an item in the ArrayCollection
  - Variable name item traditionally used, but not required
- Function returns a Boolean
- Automatically loops through each element of the ArrayCollection
  - True evaluation result indicates to include the element in the new view
  - False evaluation result indicates to not include the element in the new view
- Add your own conditional logic that checks data

Using the function

- Apply the function to the filterFunction property of an ArrayCollection
- Call the refresh() method on the ArrayCollection to apply the filter (or sort) resulting in a new view of the dataProvider
  - Returns true if successful
  - Returns false if not

Removing the filter

- If a user interaction should put all items back into the ArrayCollection
  - Best practice to set the filterFunction property to null rather than have the function called and return false for every item
Code example

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="simpletestCat.send();simp
letestFood.send()" >

<mx:Script>
<![CDATA[
import mx.collections.IViewCursor;
import
mx.controls.dataGridClasses.DataGridColumn;
import mx.collections:ArrayCollection;

private var cursor:IViewCursor;
[Bindable]
private var categoryList:ArrayCollection;
[Bindable]
private var foodList:ArrayCollection;

private function filterByCategory(event:Event):void
{
  if(event.target.selectedIndex==0)
  {
    foodList.filterFunction = null;
  }
  else
  {
    foodList.filterFunction = catFilter;
  }
  foodList.refresh();
}

private function catFilter(item:Object):Boolean
{
  return item.category_name ==
  categorySelect.selectedItem.category_name;
}
]]>
</mx:Script>

<mx:HTTPService id="simpletestCat"
url="assets\categorylist.xml"
```
<mx:HTTPService id="simpletestFood"
    url="assets\foodlist.xml"
    result="foodList=simpletestFood.lastResult.list.food"/>

<mx:ComboBox id="categorySelect"
    dataProvider="{categoryList}" labelField="category_name"
    change="filterByCategory(event)"/>

<mx:List id="foodListBox"
    dataProvider="{foodList}" labelField="product_name"
    width="200"/>

</mx:Application>
Walkthrough 8: Filtering photo data based on category

In this walkthrough, you will perform the following tasks:
- Create a filter function to compare photo categories to the category selected in a ComboBox
- Apply the filter function to an ArrayCollection

Steps

1. Open SelectPhoto.mxml.

Create and use a filtering function

2. In the Script block just below any existing functions, add a new filterByCat() function datatyped as Boolean and accept a parameter named item datatyped as Object.

   ```ActionScript
   private function filterByCat(item:Object):Boolean
   {
   }
   ```

3. In the function, return the Boolean value calculated by a comparing the CATEGORYID property of the ComboBox component’s selectedItem to the item parameter’s CATEGORY.

   ```ActionScript
   private function filterByCat(item:Object):Boolean
   {
   return (catSelect.selectedItem.CATEGORYID == item.CATEGORYID);
   }
   ```

4. In the Script block just below any existing functions, add a new filterPhotos() function datatyped as void.

   ```ActionScript
   private function filterPhotos():void
   {
   }
   ```

5. In the function, insert an if..else statement that checks if the ComboBox’s selectedIndex is equal to 0. If it is, set the
filterFunction to null. If it is not, set the filterFunction to filterByCat.

private function filterPhotos():void
{
    if (catSelect.selectedIndex==0)
    {
        photos.filterFunction=null;
    }
    else
    {
        photos.filterFunction=filterByCat;
    }
}

6. As the last line of code in the function, refresh the photos ArrayCollection.

private function filterPhotos():void
{
    if (catSelect.selectedIndex==0)
    {
        photos.filterFunction=null;
    }
    else
    {
        photos.filterFunction=filterByCat;
    }
    photos.refresh();
}

7. Add a change event handler to the catSelect ComboBox that calls filterPhotos().

    change="filterPhotos()"

8. Run the application.

    You should be able to filter on category by selecting a Category from the ComboBox control.
Using custom events to dispatch data from a component

The problem of using bindings

- Up to now, you have used bindings to obtain data from a component
- Need to be invasive into a component to know the inner details
- If the code in that component changes, reliant code will break (such as a changed ID for an object)
- An example of this is depicted below

Figure 22: Reaching into a component to retrieve data

- A much cleaner approach would be to let the component dispatch an event, just like a Button dispatches a click event
Creating loosely coupled components

- Components should be created as black boxes
- They do not have direct knowledge of any other components’ inner workings
- What data they require is sent in through binding variables to component attributes
- Data to be returned in this same manner
- Enables easier reusability and maintenance of components
- Requires an event model
Creating custom events

- Five steps to create a custom event
  1. Declare the event using the [Event] Metadata tag
  2. Create an event object
  3. Dispatch the event
  4. Create an event handler to handle the event
  5. Use the data

GOOD WAY: Rather than explicitly binding component elements to each other by name, create custom events to trigger and handle the interaction.

1. A user- or system-triggered event dispatches a custom event
2. The event object with data from the dispatching component is created
3. The main application handles the custom event and has access to the custom event object
4. The data can be used within the main application or passed to another component for use

**Figure 23: Dispatching and handling a custom event**

Declare the event

- A component needs to declare every event it can dispatch
- Subclasses can dispatch events inherited from their super classes
- With the declaration of an event, a name and an event object are specified
- In the following code block:
  - The name attribute specifies the name of the event
  - The type attribute specifies the class to instantiate
    - When the default event class is used, the type attribute can be omitted
    - When a custom event object is used, it should be a subclass of the flash.events.Event class

```xml
<mx:Metadata>
  [Event(name="eventName", type="flash.events.Event")]
</mx:Metadata>
```
Dispatch the event

- You dispatch an event using the `dispatchEvent()` method

  ```javascript
  object.dispatchEvent(event:Event):Boolean
  ```

- An event object must be dispatched with the event
- You can dispatch any event using `dispatchEvent()`, not just custom events

Example

```xml
<?xml version="1.0" encoding="utf-8"?>
  <mx:Metadata>
    [Event(name="myCustomEvent", type="flash.events.Event")]
  </mx:Metadata>

  <mx:Script>
    <![CDATA[
      private function clickHandler():void {
        var eventObj:Event = new Event("myCustomEvent");
        dispatchEvent(eventObj);
      }
    ]]>]
  </mx:Script>

  <mx:Button click="clickHandler()" />
</mx:VBox>
```

Create an event handler to handle the event

- A custom component that dispatches a custom event now has an event just like standard Flex components
- Handle that event just like other Flex components
- In the code below, the Button's `click` event is handled by calling the `clickHandler()` event handler

```xml
<mx:Button click="clickHandler()" />
```
In the code below, the `customEvent` event of the custom component in this example calls the `customEventHandler()` event handler.

- Syntax just the same as used by a built-in component handling a built-in event:

  ```xml
  <mx:MyComponent
      myCustomEvent="customEventHandler()" />
  ```

**Complete example of custom event handling**

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml">

<mx:Script>
  <![CDATA[
    private function customEventHandler():void
    {
      theLabel.text="button inside custom component is clicked";
    }
  ]]>
</mx:Script>

<mx:MyComponent
    myCustomEvent="customEventHandler()" />

<mx:Label id="theLabel"/>

</mx:Application>
```
Demonstration 3: Creating, dispatching and handling a custom event

In this demonstration, your instructor will perform the following tasks:

- Create a custom event for the ChooseMenu component.
- Dispatch the custom event when a menu is selected.
- Create an event handler in the CustomEvents application to handle the custom event.
- Populate a property with text.
- Pass the property to another custom component.

Figure 24: Custom event flow in the walkthrough
**Steps**

1. Open the file `CustEvMainApp_demo4.mxml` from the `CustomEventsDemos` project.
2. Examine the files used to dispatch the custom event.
Sending data with a custom event

- Often you want to send data with the custom event
- This is done through creating custom event classes

Identifying the need for a custom event class

- So far we have used the custom event as a notifier that something happened in the component
- Sometimes you need to send data with the custom event
  - For example, the CustomEventClass needs to send the name of the selected menu with the custom event
  - The flash.events.Event class does not support adding properties to it
- Therefore, you need to create a subclass of the Event class and add properties to it

Identifying what a custom event object must do

1. Create a subclass that extends the Event class
2. Add properties to the subclass
3. Create the custom event class' constructor
   a. Subclass the `super()` method of the Event class, passing the event type to it
4. Override the `clone()` method of the Event class

Creating a custom event class

- When you create a custom event class, it should extend the flash.events.Event class

```java
package events {
    import flash.events.Event;

    public class CustomEventClass extends Event {
    }
}
```
Adding properties

- You can now add properties for the data that you want to send with the custom event

  ```actionscript
  public var custEvProperty:String;
  ```

Creating the constructor

- The constructor should do at least 2 things:
  - Call `super()`
  - The constructor of the Event Class is defined to take the type as an argument, so you can pass the type property directly to the `super` class to set it
  - Populate the properties you added
    - Your properties can be of any built-in types such as Strings, Numbers etc.
    - Can also be typed as a custom class

Constructor arguments

- The constructor takes two arguments
  - The data you want to pass with the custom event
  - The type

  ```actionscript
  public function customEventClass(
      custEvPropertyParameter:String,
      type:String
  ){
      super(type)
      this.custEvProperty=custEvPropertyParameter;
  }
  ```

  Tip: The `this` prefix is not required; It would be if the parameter name matched the class property name, which is often the case

Overriding the clone method

- Override the `clone()` method of the super class
  - `clone()` method returns a new instance of your custom event class
  - Required to support event bubbling, a topic not covered in this course

  ```actionscript
  override public function clone():Event
  {
      return new CustomEventClass(
          custEvProperty,type);
  }
  ```
Using the custom event class with the custom event

- To use the custom event class
  - Declare the custom event using Metadata
  - Dispatch the custom event

Declare using Metadata

- In the Metadata section, you declare your custom event
  - Use the name argument to name the custom event
  - Use the type argument to datatype the custom event
    - The type will refer to a custom event class

```xml
<mx:Metadata>
  [Event(name="custEvName", type="events.CustomEventClass")]
</mx:Metadata>
```

Dispatch the custom event

- To dispatch the custom event
  1. Create an instance of your custom event class
  2. Populate the properties
  3. Send it with the dispatched event

Example

```javascript
private function clickHandler():void
{
    var eventObj:CustomEventClass = new CustomEventClass(customPropertyValue, "custEvName");
    dispatchEvent(eventObj);
}
```

- where
  - CustomEventClass is the custom event that you created
  - customPropertyValue is the value you want to use to populate your custom property
  - customEvName is the custom event name defined in the Metadata tag
Demonstration 4: Creating a custom event object and sending it with the custom event

In this demonstration, your instructor will perform the following tasks:

- Create a custom event object Class.
- Populate a custom property of the custom event object.
- Send the custom event object with the custom event.
- Use the custom property in the event handler.

Steps

1. Open the file CustEvMainApp_wt1.mxml from the CustomEventsDemos project.
2. Examine the files used to dispatch the custom event with data associated with it.
Walkthrough 9: Dispatching photo data with a custom event

In this walkthrough, you will perform the following tasks:
- Create a custom event class
- Define and dispatch a custom event
- Handle the custom event
- Display the photo dispatched in the custom event

Steps

Create a custom event class

1. Right click on the AD003H project and create a new folder named events.
2. Right click on the events folder and select New > ActionScript Class.
3. Be sure the Package is events.
4. Set the Name to be PhotoEvent.
5. Click the Browse button next to the Superclass value and select Event - flash.events.
6. Click Finish.
7. Below the existing import statement also import valueObects.Photo.
8. In the class definition, create a public variable named selectedPhoto datatyped as Photo.

```actionscript
package events {
    import flash.events.Event;
    import valueObects.Photo;

    public class PhotoEvent extends Event {
        public var selectedPhoto:Photo;
    }
}
```
9. Below the variable declaration, create the class’ constructor. It should accept two parameters, the first named selectedPhoto datatyped as Photo and the second named type datatyped as String.

```actionscript
public function PhotoEvent(selectedPhoto:Photo, type:String)
{
}
```
10. In the constructor, call the `Event`'s constructor (the parent of this class) using the `super()` method passing the `type` parameter.

```
super(type);
```

11. Next assign the class' `selectedPhoto` property the `selectedPhoto` parameter.

```
this.selectedPhoto=selectedPhoto;
```

12. Create another public function that overrides the `clone` method, datatyped as `Event`.

```
override public function clone():Event
{
}
```

13. In the function return a new instance of the `PhotoEvent` custom event object, passing `selectedPhoto` and `type` as parameters.

```
return new PhotoEvent(selectedPhoto,type);
```

14. Your class should appear as follows.

```
package events
{
    import flash.events.Event;
    import valueObjects.Photo;

    public class PhotoEvent extends Event
    {
        public var selectedPhoto:Photo;

        public function PhotoEvent(selectedPhoto:Photo,type:String)
        {
            super(type);
            this.selectedPhoto=selectedPhoto;
        }
        override public function clone():Event
        {
            return new PhotoEvent(selectedPhoto,type);
        }
    }
}
```
Define the custom event

15. Open SelectPhoto.mxml.
16. At the top of the Script block import the `events.PhotoEvent` class.
17. At the top of the component just below the VBox and above the `<mx:Script>` block, declare a custom event, using `<mx:Metadata>`. Use `photoSelected` as the name for the custom event, and use the `events.PhotoEvent` class for the type:

```xml
<mx:Metadata>
    [Event(name="photoSelected",
    type="events.PhotoEvent")]
</mx:Metadata>
```

Add an event to the Image

18. Add a click event to the Image tag. The event should call the `dispatchPhoto()` function and pass `event.currentTarget.getRepeaterItem()`.

```xml
<mx:Image source="assets/{photoRepeater.currentItem.FILENAME}"
    width="80" height="80"
    click="dispatchPhoto(event.currentTarget.getRepeaterItem())"/>
```

Dispatch the custom event object

19. Locate the empty `dispatchPhoto()` function. Change the parameter is accepts to be named `selectedPhoto` datatyped as `Photo`.

```xml
private function dispatchPhoto(selectedPhoto:Photo):void
{
}
```

20. In the function, set a variable local to the function named `eventObj` datatyped as `PhotoEvent` equal to an instantiation of the `PhotoEvent` class. Pass the `selectedPhoto` and the String `photoSelected` as parameters.

```xml
var eventObj:PhotoEvent = new PhotoEvent
    (selectedPhoto,"photoSelected");
```

21. As the last line of code in the function, use the `dispatchEvent()` method to dispatch the `eventObj` object.
22. Your function should appear as follows.

```ActionScript
private function dispatchPhoto(selectedPhoto:Photo):void
{
    var eventObj:PhotoEvent=new PhotoEvent(selectedPhoto,"photoSelected");
    dispatchEvent(eventObj);
}
```

**Handle the custom event**

23. Open FStop.mxml.
24. At the top of the Script block, import the `events.PhotoEvent` class.
25. Locate the instantiation of the SelectPhoto component (use the Outline view for help).
26. In the component, handle the `photoSelected` custom event and call the `photoSelectedHandler` passing the `event` object as a parameter.

```ActionScript
photoSelected="photoSelectedHandler(event)"
```
27. In the Script block just below any existing functions, create a private function named `photoSelectedHandler()` datatyped as `void`. The function should accept one parameter named `event` datatyped as `PhotoEvent`.
28. Place a breakpoint on the closing brace of the function.
29. Debug the application and click on a photo.
30. In the debugger be sure `event.selectedPhoto` has the proper values and is datatyped correctly as a `Photo` object.

![Figure 25: The selected photo from the custom event](image)

31. Terminate the debugging session and remove the breakpoint.
32. In the Script block just below any other variable declarations, create a bindable, private variable named `selectedPhoto` datatyped as `Photo`.

```ActionScript
private var selectedPhoto:Photo;
```
Note: At this point you can remove the dummy variable declared with the statement
private var dummyPhoto:Photo. You now have a variable you need datatyped as Photo.

33. In the photoSelectedHandler() function, assign the
    selectedPhoto variable the value of event.selectedPhoto.

    selectedPhoto=event.selectedPhoto;

**Display the selected photo**

34. Open DisplayPhoto.mxml.
35. In the Script block, create a bindable, public variable named
    selectedPhoto datatyped as Photo.

    import valueObjects.Photo;

    [Bindable]
    public var selectedPhoto:Photo;

36. Below the Script block, display the photographer’s name, the description
    and the photo.

    <mx:Label
text="{selectedPhoto.PHOTOGRAPHER}"/>
    <mx:Label text="{selectedPhoto.DESC}"/>
    <mx:Image
        source="assets/{selectedPhoto.FILENAME}"/>

37. Return to FStop.mxml and locate the instantiation of the DisplayPhoto
    component.
38. Bind the selectedPhoto variable to the selectedPhoto property of
    DisplayPhoto.
39. Run the application.

    You should see when you click on a photo the details of the photo are displayed above the thumbnails.
Using view states

- Important reasons why Flash and Flex are powerful for application development:
  - To display new content in an HTML page, you would create another page with the new content.
  - HTML works in discrete pages.
  - DHTML allows for more dynamic effects, but is complex to program and maintain.
  - Flash applications can be unified applications, which means that the content in the movie can move around or redraw without the need to refresh the page in the browser.
  - Traditionally Flash applications were managed:
    - Manually on the timeline
    - Programmatically
  - Flex 2 introduces view states, which allow developers to easily manage changes in presentation layouts.
    - Components can persist across different tasks.
    - Components can be reconfigure for use in different contexts.
    - Can be created and managed within Flex Builder with or without writing code.

Introducing view states

- View states define distinct presentation layouts within the same application.
- Each distinct layout is called a state.
- You use user or system events to change states.

Figure 26: Flash and Flex allow you to create dynamic applications that can change their presentation layouts without refreshing the browser window.
View states examples

Example #1

E-commerce Product Thumbnail

Figure 27: An example of how view states are used in an e-commerce application.

Example #2

Product Searching Drill-Down

Figure 28: An example of how view states are used in a search interface.

Creating view states

1. Create a base state, which is the default layout of the application or custom component.
2. Modify the base state or another state to create additional states.
   • Can add, modify or delete components.
   • Can modify component properties.
   • Can change styles.
3. Create events to trigger switching states.

Benefits of using view states

• Flex manages the switching of states.
- Components can be shared across multiple view states.
- Components can be added, removed, or modified easily.

Creating view states with Design mode

- Draw-backs to complex unified application development:
  - Can be complex to build and maintain.
  - Can be hard for a designer to prototype the states due to complex programming.
- Flex Builder 2
  - Allows designers or developers to quickly and easily create application states.
  - Provides a view to manage states.
- Creating the base state
  - Create the base state using Design mode or Source mode as usual.
  - Create an element that can trigger the state change.

![Figure 29: Users trigger the state change.](image1)

- Creating a new state
  - Switch to Design mode in Flex Builder.
  - Right-click on the Base state indicator in the States view (Window > Show View > States) and select New State.

![Figure 30: Create a new state using the States view in Flex Builder 2.](image2)
In the **New State** dialog box, name the new state and base it on the base state or another state.

![New State dialog box](image)

*Figure 31: You choose a state on which to base your new state.*

**Note:** You can set the new state to be the state that is displayed when the application starts by checking the **Set as start state** checkbox.

- Make sure the new state is selected in the **States** view and then start adding elements to the application in **Design** mode.
- You can add, modify or delete components.
- You can modify styles.
- The changes are recorded as part of the new state.
- States are mutually exclusive – only one can be displayed at a time.
- You can base a new state on a state other than the base state. You should do this in a case where a state is shared by multiple states. For instance, the same navigation bar would be displayed by multiple states. These states should be based on the navigation bar state.

## Controlling view states

- **State implementation details**
  - To change states, set the **currentState** property.
  - States can be derived from other states as well as from the base state.
  - Flex figures out everything that needs to change between any two states.
  - The **Base** state doesn’t have to be the initial state.
  - Use an event to trigger a state change.
  - Because Flex always starts from the base view state when it applies a new view state, it is more efficient to use the most commonly used state as the base state.

- **Setting **currentState**
  - Setting the **currentState** property of the **Application** tag sets the initial state to one other than the base state.
  - A component’s **currentState** property specifies its view state.
  - To specify the base view state, set **currentState = ''**.
In the event handler of a component, set the `currentState` property equal to the name of a view state.

```xml
<mx:Button id="b1" label="Add a Button"
    click="currentState='newButton'"/>
<mx:Button id="b2" label="Remove Added Button"
    click="currentState=''"/>
```

In Flex Builder, select the component that will trigger the event and set its Click property using the Common properties section of the Flex Properties view.

*Figure 32: Declare which state to switch to when the user clicks by setting its name in the `currentState` property.*
Walkthrough 10: Creating a view state

In this walkthrough, you will perform the following tasks:
- Create a new view state
- Change property values and add children in the new view state
- Set the currentState property to move between the view states

Steps

1. Open FStop.mxml.
2. Set the visible property on the DisplayPhoto component to false.

Create the new view state

3. Move into Design mode.
4. Create a new state named photoDetailVS and be sure it is selected.
5. From the Outline view, select the SelectPhoto custom component and change its visible property to false.
6. From the Outline view, select the DisplayPhoto custom component and change its visible property to true, and be sure its x and y property values are 240 and 147.
7. Still in the photoDetailVS, drag a Button from the Components view and place it just below the image at the top of the page. Use the guides to center the Button.
8. Change the label on the Button to read Return to Thumbnails.
9. Click between base state and the photoDetailVS state and note the differences.

Set the currentState property to move between view states

10. Select the Button in the photoDetailVS.
11. In the Flex Properties view, set the On click entry to currentState='' to return to the base state.
12. Return to Source mode.
13. Locate the photoSelectedHandler() function.
14. As the first line of the function, set the `currentState` property equal to `photoDetailVS`.

```flex
private function photoSelectedHandler(event:PhotoEvent):void {
    currentState="photoDetailVS";
    selectedPhoto=event.selectedPhoto;
}
```

15. Run the application.
You should be able to click on a thumbnail image and the new view state will display the image details. Click on the Button to return to the base state to see the thumbnails again.

16. Open DisplayPhoto.mxml and add a `color` attribute to both Labels set equal to `#F8E5BB`.

```xml
<Label color="#F8E5BB"/>
<Label color="#F8E5BB"/>
```

17. Run the application.
You should be able to click on a thumbnail image and the new view state will display the image details. The text will be easier to see.
Using Transitions

- Use transitions and effects to form a smooth transition from one state to another
- A transition is one or more effects grouped together to play when a view state change occurs.
- A transition is different than a behavior.
  - A behavior applies effects to components.
  - A transition applies effects to view states.
- You can apply one or more effects to one or more components in a view state, and you are not limited to the same effects when expanding or collapsing a state.
- Conditional transitions are called effect filters and can apply different effects to the same target based on the condition. Effect filters are outside the scope of this module.
- You use the Transition class to create a transition. The following table defines the properties of the Transition class:

<table>
<thead>
<tr>
<th>Transition class property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromState</td>
<td>A String that specifies the view state that you are changing from when you apply the transition. The default value is an asterisk (*) which means any view state.</td>
</tr>
<tr>
<td>toState</td>
<td>A String that specifies the view state that you are changing to when you apply the transition. The default value is an asterisk (*) which means any view state.</td>
</tr>
<tr>
<td>effect</td>
<td>The Effect object to play when you apply the transition. Typically, this is a composite effect, such as the Parallel or Sequence effect, that contains multiple effects.</td>
</tr>
</tbody>
</table>
Implementing transitions

- Surround one or more Transition tags with the transitions tag.

```xml
<mx:transitions>
  <mx:Transition id="myTransition"
      fromState="state1" toState="state2">
      [...]
  </mx:Transition>
  <mx:Transition id="myTransition2"
      fromState="state2" toState="state3">
      [...]
  </mx:Transition>
  <mx:Transition id="myTransition3"
      fromState="*" toState="*">
      [...]
  </mx:Transition>
</mx:transitions>
```

- Place the desired effects defined earlier in the section on Behaviors within the Transition tag.
  - use the target property of the effect to declare the one component to be animated.
  - use the targets property of the effect to declare multiple components to be animated.

```xml
<mx:transitions>
  <mx:Transition id="myTransition2"
      fromState="state2" toState="state3">
      <mx:WipeLeft duration="400" />
  </mx:Transition>
</mx:transitions>
```

- The Parallel and Sequence tags must be used to define multiple effects within the transition.
  - Use the target property of the effect instance or the Parallel and Sequence tags if there is only one target.

```xml
target="{panel1}" 
```

- Use the targets property of the effect instance or the Parallel and Sequence tags if there is more than one target.

```xml
targets="[{panel1, panel2, panel3}]"
```
SetPropertyAction

- The SetPropertyAction class defines an action effect that changes a property value of an object involved in a transition
  - Properties
    - target: The object whose property you wish to change
    - name: The name of the property to change
    - value: The new value for the changed property
Walkthrough 11: Adding transition

In this walkthrough, you will perform the following tasks:
- Create two transitions to be used when moving between view states
- Use the Iris effect
- Use SetPropertyAction to control the visible property of custom components during a transition

Steps

1. Open FStop.mxml.

Create the first transition

2. Insert an `<mx:transitions>` tag block just below the closing `</mx:states>` tag.
4. Set the `id` of the Transition to `leaveBase`, the `fromState` to the base state, and the `toState` to `photoDetailVS`.
5. Nest an `<mx:Sequence>` tag block in the `<mx:Transition>` tag block.
6. In the Sequence block, insert an `<mx:Iris>` effect to make the thumbnails disappear. Bind the `target` to `selectPhoto`, set the `duration` to `1000` and `showTarget` to `false`.

```xml
<mx:Iris target="{selectPhoto}" duration="1000" showTarget="false"/>
```

7. Use an `<mx:SetPropertyAction>` tag to set the `selectPhoto` component's `visible` property to `false`.

```xml
<mx:SetPropertyAction target="{selectPhoto}" name="visible" value="false"/>
```

8. Use another `<mx:SetPropertyAction>` tag to set the `displayPhoto` component's `visible` property to `true`.

```xml
<mx:SetPropertyAction target="{displayPhoto}" name="visible" value="true"/>
```
9. As the last tag in the Sequence, add another `<mx:Iris>` effect to make the photo details appear. Bind the target to `displayPhoto`, set the duration to 1000 and `showTarget` to true.

```xml
<mx:Iris target="{displayPhoto}" duration="1000" showTarget="true"/>
```

10. Check to be sure your `<mx:transition>` block appears as follows.

```xml
<mx:transitions>
  <mx:Transition id="leaveBase"
    fromState=""
    toState="photoDetailVS">
    <mx:Sequence>
      <mx:Iris target="{selectPhoto}" duration="1000"
       showTarget="false"/>
      <mx:SetPropertyAction
         target="{selectPhoto}"
         name="visible"
         value="false"/>
      <mx:SetPropertyAction
         target="{displayPhoto}"
         name="visible"
         value="true"/>
      <mx:Iris target="{displayPhoto}" duration="1000"
       showTarget="true"/>
    </mx:Sequence>
  </mx:Transition>
</mx:transitions>
```

11. Run the application.

You should see the transition working when you click on a thumbnail. When you click on the Button to return to the thumbnails no transition yet exists.

12. Copy the existing `<mx:Transition>` block and paste another copy directly below itself, still in the `<mx:transitions>` block.
13. Alter the copied Transition to have the Iris effects work when returning to the base state.

```xml
<mx:Transition id="toBase"
    fromState="photoDetailVS"
    toState="">
    <mx:Sequence>
        <mx:Iris target="{displayPhoto}"
            duration="1000"
            showTarget="false"/>
        <mx:SetPropertyAction
            target="{displayPhoto}"
            name="visible"
            value="false"/>
        <mx:SetPropertyAction
            target="{selectPhoto}"
            name="visible"
            value="true"/>
        <mx:Iris target="{selectPhoto}"
            duration="1000"
            showTarget="true"/>
    </mx:Sequence>
</mx:Transition>
```

14. Run the application.

You should see the transitions work in both directions.