Flex in a Week, Flex 4.5

Video 2.02: Understanding the event object and bubbling

You learned in the last video that the Flex framework is an event-driven environment, where actions are performed based on system or user events.

In this video, you will learn about the event object that is automatically passed around with an event instance and contains useful information for you to capture.

You will also learn about event propagation, which includes the capturing, targeting and bubbling phases of event handling.

In the last exercise, you created a function that handles the user click on the DateChooser controls.

Initially you handled the click inline in the MXML tag, but then moved the code into a function so that you could reuse it in both the pickup date and return date calendars.

When I click on the pickupDate calendar instance, you can see that the Alert dialog appears as expected with information about the date I selected.

When I refresh the application and then click on the returnDate calendar instance, you will see an ActionScript error message appear.

Here is the code for the event handler.

You can see that the Alert message explicitly displays the selectedDate property from the pickupDate control.

This means when I click the returnDate, there is no selectedDate in the pickupDate, which will make the code fail and the error message appear.

In this video, I will show you how to extract the instance id for the calendar that was clicked from the event object.

This is the same application, but with the changes I will implement in this video.

You can see that when I click the returnDate, the proper message appears about the date that I selected.

This still works for the start date as well.

I will also use information in the event object to add some validation to the controls.

For instance, if the user clicks an end date that occurs before the start date, then he will see this message.

In this video you will learn about the event object that is automatically instantiated by every Flex event and how to use Flash Builder's Debugging perspective to view the properties of the object.

I will also introduce you to the Event class, its subclasses, and event propagation.

Every time an event is fired, an event object is created.

Each event is an instance of the Flex framework's Event class or one of its subclasses and stores information about the specific event.

Some of the properties in the event object are standard properties for all events and others are specific to the event that occurs.

Passing the event object to the event listener is optional.

If you do choose to use the event object, you pass it as an argument to the event handler.

This is the starter file for this video and associated exercise.

I have located the pickupDate instance and am placing my cursor between the parentheses of the dateChangeHandler() function call.

I am typing the word event to pass the event object to the handler.

Of course I now need to update the function to accept the argument.

I am scrolling up to the Script block to locate the dateChangeHandler() function. I am adding the event object as an argument for the function.

Notice, that by convention, I am naming it "event".

I am also data typing it to the Event class, which I do not need to import.

Remember that both the pickupDate and returnDate instances use the same change event handler.

I am updating the returnDate instance with an event object as well.

As I mentioned before, every event object is an instance of the Event class and contains certain properties that exist for every event.

The event type is a string value that tells us what type of event was fired; for example, "click", when the user clicks on a control.

The target property is an object with information about the component instance that broadcasted, or dispatched, the event.

This target object includes the id property, which is the name of the target; in my example, this would return the instance names for either of the DateChooser controls, which are pickupDate or returnDate.

Within the event object, some properties are specific to the event being broadcasted.

In later videos, you will learn how to extend the Event class to create your own properties to pass in the event object.

I can use the Flash Builder Debugging tools to view the event object.

I am placing a debugging breakpoint in the event handler by double-clicking in the Editor marker bar on a line of ActionScript.

If you don't have line numbers turned on, you can turn them on by right-clicking in the margin and selecting Show Line Numbers.

This makes it easier to see where to add the breakpoint.

A breakpoint will temporarily suspend the application code so that I can inspect it.

I have placed the breakpoint on the closing curly brace of the event handler, because I want the code to run to this point before it moves into the Debugging perspective.

Instead of running the application, I am clicking on the Debug button.

Flash Builder prompts me to save the file and then will compile the SWF and open it in the browser, using the special debug version of the Flash Player.

The application displays normally in the browser, but when I click on a date in either of the DateChooser instances, the application will run the event handler and hit the breakpoint that I defined.

This will pause the application in the browser and return to Flash Builder, prompting me to switch to the Flash Debugging Perspective.

Once in the new perspective, I can double-click on the Variables tab to maximize it.

Here is the event object and its properties.

Notice that the event type is "change" and that the target object is the DateChooser control.

Also notice that the event object is an instance of the CalendarLayoutChangeEvent subclass of the Event class.

I am double-clicking on the Variables tab again and then clicking on the Terminate button to stop the debugging session.

This will also close the browser.

Lastly, I will switch back to the Flash perspective to view my code.

This is the Event class documentation in the Language Reference.

You can see all of the Event subclasses in the flash.events package.

Notice the CalendarLayoutChangeEvent listed here.

I can click on it to view more information about this subclass.

When you declare an event object, you can declare it as an instance of the Event class or you can specify a subclass.

Four examples of subclasses are MouseEvent, DropDownEvent, MoveEvent and CalendarLayoutChangeEvent.

Subclasses are mostly defined in three packages.

The spark.events package defines event classes that are specific to Spark.

The mx.events package defines event classes that are specific to Flex framework components in the MX library.

The flash.events package describes events that are not unique to Flex but are instead defined by Flash Player.

Note that you will not need to import any classes that are in the flash.events package.

Each Event subclass provides additional properties and methods that are unique, and you will sometimes want to use the more specific event type to access them.

Additionally, using subclasses results in stricter datatyping, faster runtime performance and compile-time type checking.

I am updating the data type for the event object in the handler to reflect the specific event subclass, which is CalendarLayoutChangeEvent class.

Note that the import statement was not automatically included since I typed the class name by hand.

However, if I delete the colon and type it again, I do get the Flash Builder code hinting and as I start to type the class, it does appear and I can select it.

The import statement does get added with the CalendarLayoutChangeEvent class.

Always use Flash Builder’s code assist tool if you can, since it has some very convenient supportive features.

You may have noticed that the event object contains two objects named target and currentTarget.

Both objects will often have the same value, but each has a separate purpose.

Later in this video, I will discuss event propagation, which will help you understand the difference between these two objects.

For now notice that the target object has a property named id, which contains the id property of the target object that was clicked.

In this case, I clicked on the pickupDate instance
I will use the target property of the event object to dynamically determine which instance of the DateChooser control was changed.

I am replacing the hard-coded pickupDate instance reference with the event.target property.

When I save the file and run the application, you will see that the Alert dialog message appears with an appropriate date for both DateChooser instances.

Notice that if you type event.target.selectedDate, Flash Builder will not automatically give you the toDateString() code hinting.

This is because Flash Builder does not know that this value is a Date instance.

You can cast event.target.selectedDate as a Date instance, adding parentheses.

Now when you type a period, the toDateString() method will now appear in code hinting.

The application will run as before.

In this section of video, I will make the application validate whether the return date is after the pickup date.

Keep in mind that the topic of this video is the event object, not validation.

The validation that I implement here is simple and meant only to illustrate the use of the event object.

You can, of course, enhance this validation with your own code.

You will also learn more about validation in a later video.

Below the Alert.show() method in the dateChangeHandler() function, I am pressing CTRL+Space twice to show code templates using the content assist tool and selecting the if template to create a conditional if statement.

I am evaluating two separate statements so I am placing two sets of parentheses in the if statement separated by double ampersands, which denote an AND statement.

This means that both conditions must be true for the entire if statement to be true.

In the first condition, I am checking whether the value of the event.target.id, which is the id of the selected DateChooser instance, is equal to returnDate.

Note that in this previous reference for event.target, I am referencing the selectedDate property because I am displaying the date selected in the chooser.

In this conditional statement, I am literally checking for the instance name of the DateChooser instance, so I am referencing the id property.

I am explicitly checking for the string value of the id so I am placing single quotes around returnDate.

In the second condition, I am checking whether the pickupDate.selectedDate is greater than the returnDate.selectedDate.

In other words, this condition will be true if the the user just selected a date in the returnDate instance and the pickup date is after the return date.

In the condition, I am calling the Alert.show() method to display the message “Pickup date must be scheduled before return date.”

I am saving the file and running the application.

First I will select a date from the pickupDate instance.

You can see that the message that displays the chosen date appears as expected.

When I now select a return date from the returnDate instance that is before the pickup date, you will see the Alert window appear with the validation message.

I am clicking OK.

The second message appears.

Note that the validation message appears first because it is stacked on top of the first message in the application display and in the code.

In the last section of this video, I want to introduce you to the concept of event propagation.

This code shows a Button control placed inside of a Panel container, which is inside of an Application container.

So far in this video series, we have placed events, like the click event, on the controls that dispatch the event itself.

For instance, the click event is on the Button control.

However, there are times when it is more convenient to handle an event on the parent of an object, rather than on the dispatching object itself.

I am moving the click event by holding down the alt key and pressing the up arrow into the parent Panel container.

Likewise, I could handle the Button click on the Application container.

In the Flex framework's event propagation mechanism, you can choose to handle an event on the dispatching target object, in this case, the Button control, or in any of its parent ancestor containers.

Consider this representation of the code example I just discussed.

The outermost element is the Application container.

Inside the Application container is the Panel container.

Inside of the Panel container is the Button control.

Keep in mind that what I am about to illustrate will not be entirely technically accurate, but the example will give you a point of reference for understanding the concepts behind the three event propagation phases, which are: capturing, targeting and bubbling.

Let's start by turning the application to its side view.

You can see the Button control represented in yellow in the middle and surrounded by both the Panel and Application containers.
The capturing phase of event propagation is the first phase.

Before the target object is checked for event listeners, the application will first check all of the parent, or ancestor, containers for listeners that are registered to this phase.

The check will occur from the outermost ancestor to the ancestor which is the immediate parent of the target object.

From the side view, you can see that in order for a user to click on the button he will first touch the Application container.

The capturing phase will then move on to the next inner ancestor, which, in this case, is the Panel container.

Only after all ancestors have been checked for capturing phase listeners will the actual target object be reached, which is the Button.

The targeting phase of event propagation is the second phase and the one with which you are most familiar.

It is in this phase, that the application checks for listeners on the target object.

In this example, the Button will check for registered listeners for events, like the common click event.

The bubbling phase of event propagation is the third, and last, phase.

After the target object is checked for event listeners, the application will check all of the parent, or ancestor, containers for listeners that are registered to this phase.

The bubbling phase is the opposite of the capturing phase and checks for listeners from the direct parent ancestor of the target object to the outermost ancestor.

In my example, bubbling starts at the Panel container and continues outward until it reaches the main Application container.

Remember that the point of reviewing the phases of event propagation was to explain the difference between the target and currentTarget objects in the event object.

The target object always refers to the object that dispatched the event.

The currentTarget object always refers to the object that is currently being examined for listeners to that dispatched event.

Let’s return to the capturing phase.

When this phase reaches the Application container, the x currentTarget object is the Application container itself since it is the object being examined for listeners.

X The target object is the Button control since it is the object that was clicked, and so, dispatched the event.

X When the capturing phase reaches the Panel container, x the currentTarget object is the Panel container itself since it is the object being examined.

X The target object is still the Button control since it is the object that dispatched the event.

During the targeting phase, the Button is examined for event listeners and is also the event dispatcher so it is x both the target x and the currentTarget object.
This is the demo application that has a Panel container with a Button control inside of it, as in my illustration.

You can see that the Panel container has an id value of panel and the Button is called button.

In both cases, the instances names start with lowercase letters.

The Button control has a click event that calls the clickHandler() function, which displays an Alert message.

This alert message prints out the name of the target, event.target.id, and the name of the currentTarget.

When I run the application and click on the Button control, you can see that the target and currentTarget are both the button instance.

This is because the button instance contains the click handler.

Back in the code, I am moving the click event from the Button control to the Panel container.

When I save the file and run the application again, I will click on the Button control, and you can see that the target is still the button control, but the currentTarget is the panel, since that is the object that now holds the click event.

Keep this in mind as you work with events and event objects:

The target object is always the object that throws the event while the currentTarget is the object that handles the event.

In many cases, they are the same, but not always.

For your next step, work through the exercise titled “Using the event object”.