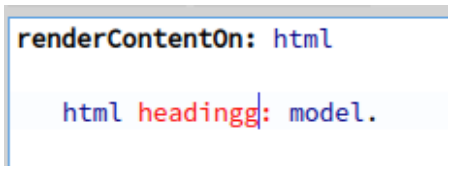
In this chapter we use the Flight Information application to learn how to use a debugger and see how Smalltalk provides uniquely powerful debugging opportunities. Then we will explore how code can be edited from a web browser and discuss code blocks, a powerful feature of Smalltalk.

1. Start Pharo if it is not running and in the System Browser select the FlightInfoComponent and the 'renderContentOn:' method. We will modify this method in such a way as to introduce an error into the code.

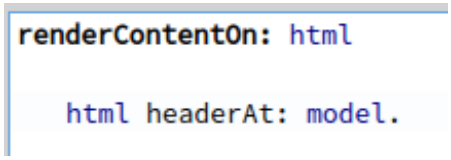
```
renderContentOn: html

   html headingg: model.
```

This changes the message sent from 'heading:' to 'headingg:' and represents a typical 'typo' that could easily be made. Note that the message is in red type, indicating that there is no object in the entire object space that understands this message.

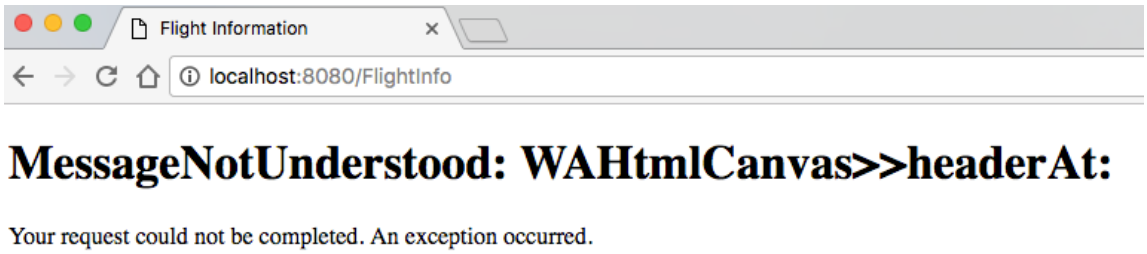


Recognizing that the method is wrong, we can change it to something else, 'headerAt:', that is recognized, but still wrong.

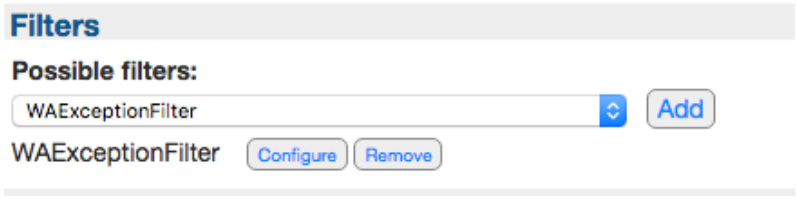


This demonstrates a problem with dynamically typed languages (such as Smalltalk). Because a variable can hold a reference to any object, we can't tell at compile time whether the receiver will understand the message we send to it. One of the advantages of a statically typed language is that this sort of error is substantially avoided. Suffice it to say that this is the topic of many language wars, and we'll let it go with the statement that Smalltalkers are almost universally happy with the flexibility offered by the dynamic environment. Of course, this means that we occasionally have to fix problems where we sent the wrong message, so here goes!

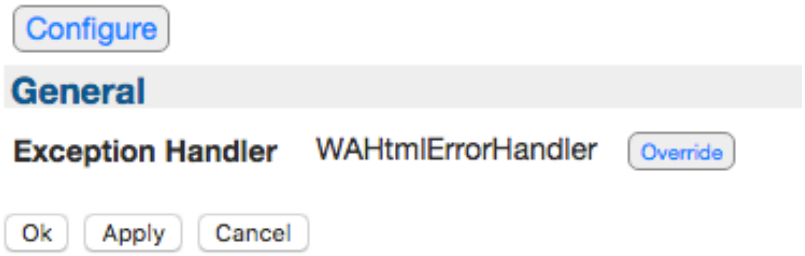
2. Return to your web browser and navigate to the 'FlightInfo' component (click the <Refresh> button if you are already there). Seaside returns an error.



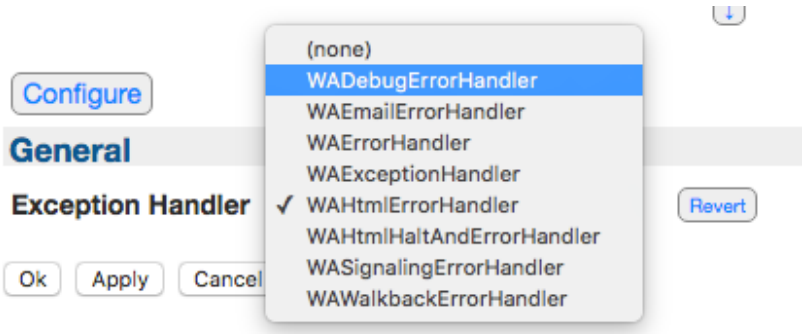
This is informative but not very helpful for debugging. In a new tab, open http://localhost:8080/config/FlightInfo and click the 'Configure' button next to 'WAExceptionFilter.'



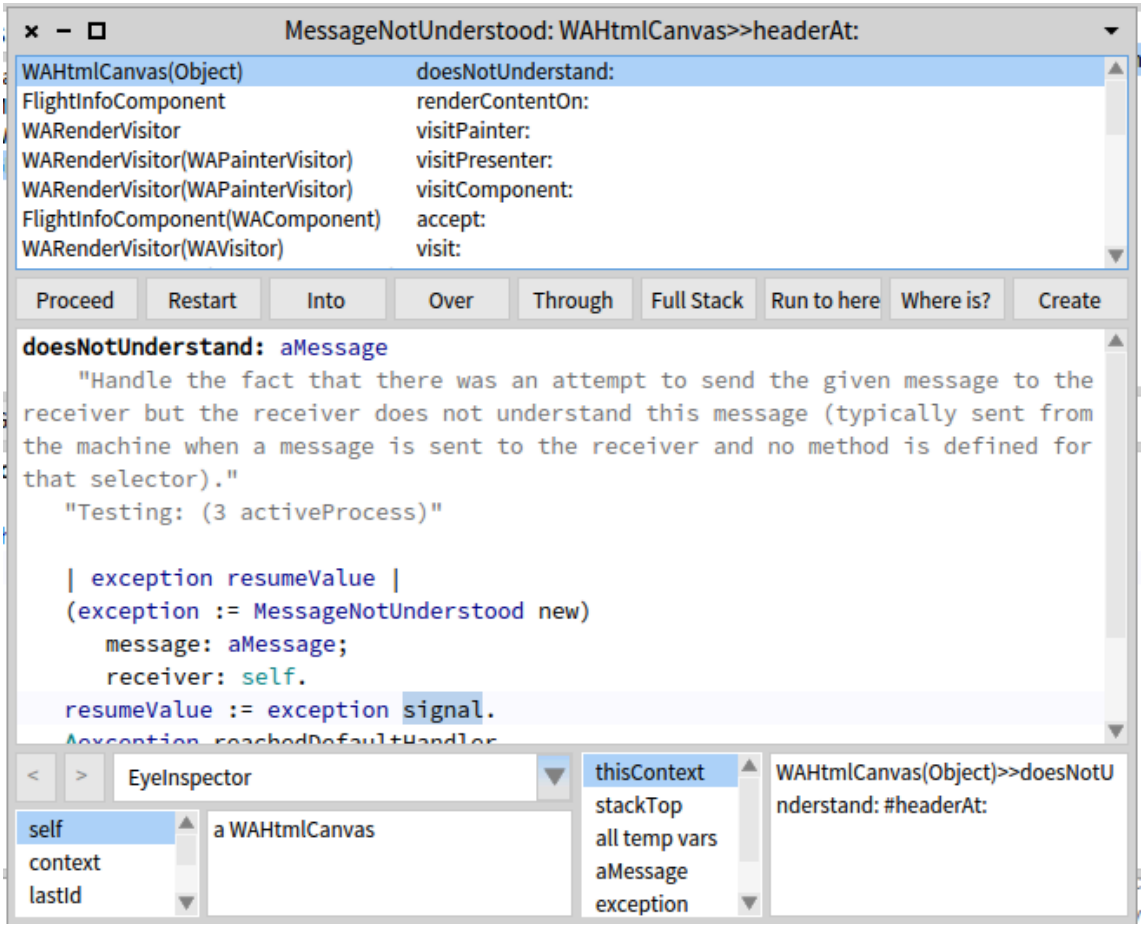
Next, click the 'Override' button next to 'WAHtmlErrorHandler'.



From the drop-down list of Exception Handlers, select 'WADebugErrorHandler' and then click the 'Apply' button.

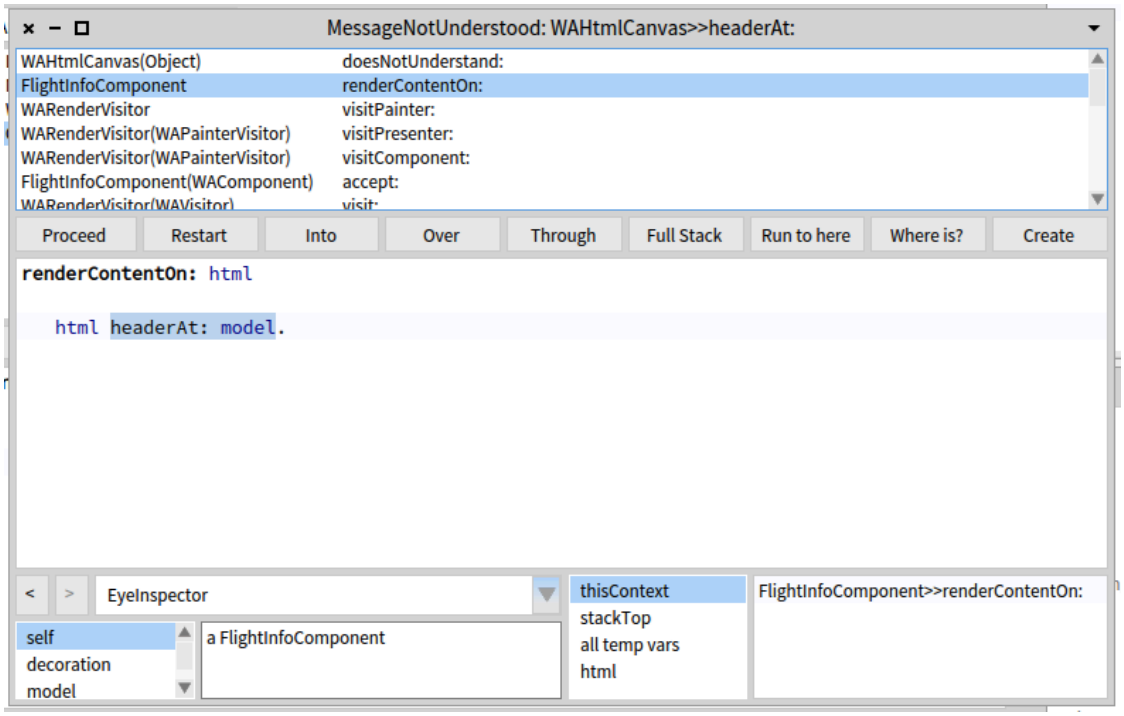


Now return to your web browser tab with the Flight Info error and click refresh. Note that the web browser just hangs. This is because Seaside got an error and has not returned any HTML to the client and it gives us an opportunity to look at Smalltalk's debugging capabilities. Switch back to Pharo and notice that a new window exists with the title 'MessageNotUnderstood'.



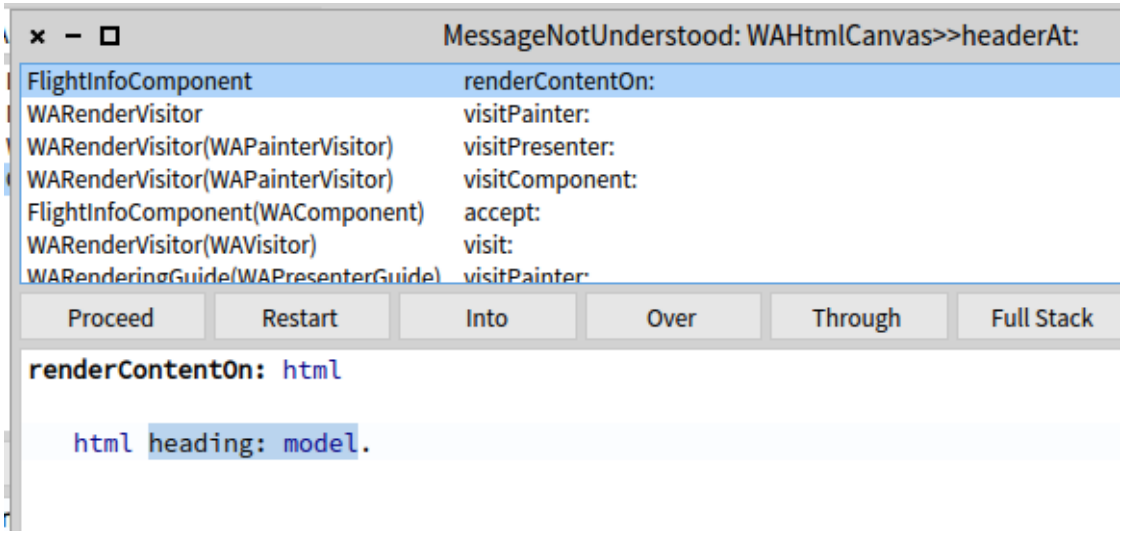
3. In the top third of this debugger window is a scrolling list of class and method names representing frames in the process stack. In the middle third we have a row of buttons followed by a text area. The buttons control the debugger and the text area shows us a method in a selected stack frame. The bottom third gives us information on the receiver and its instance variables, and the context and its temporary variables.

Click the second line in the process stack: ‘FlightInfoComponent>>renderContentOn:’.



4. At this point we can see the method is sending the message 'headerAt:' which is not understood. We should have sent the message 'heading:' to get the proper behavior.

   If this were a typical programming language, we would need to edit a text file containing the source, then recompile the code, restart the application, and apply it to the web server. Smalltalk gives us a much more powerful approach to fixing bugs. In the debugger, edit the method so that 'headerAt:' is replaced with 'heading:' and save the method (<Ctrl>+<S>). When you save the method the debugger trims the stack to this method (removing the frames above) and starts over at the beginning of this method.

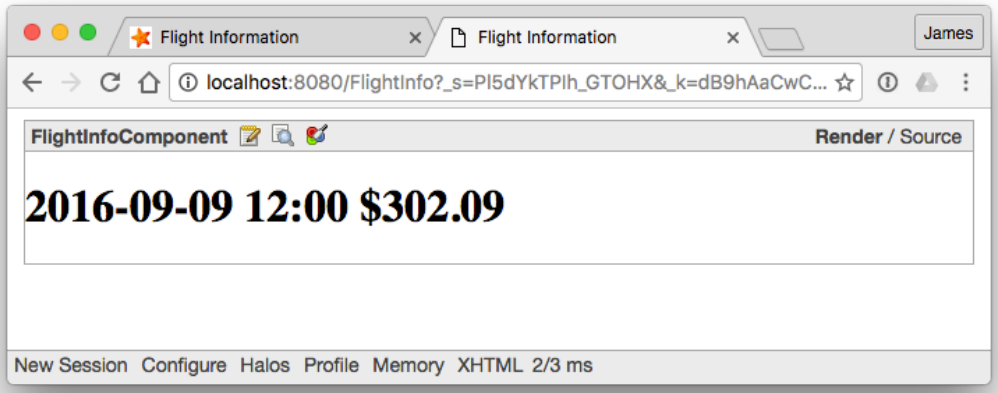


   You can click the 'Into' and 'Over' and 'Through' buttons a few times to watch the code being executed. You can also click on some of the lists at the bottom to see the values of various variables.

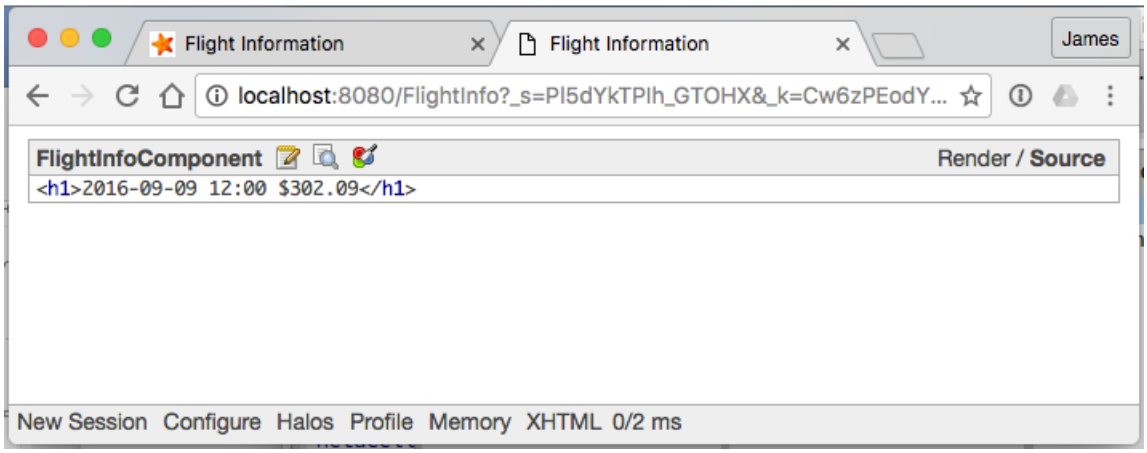
5. Once you have played a bit with the debugger, click the 'Proceed' button (the left-most of the buttons). The debugger window should close and then you can return to your web browser. The web browser should now show a page with a date, time, and dollar amount. This shows that we have fixed the problem introduced above in step #1.

   This is an example of Smalltalk's ability to do surgery on a sick patient when other environments would only give you an opportunity to do an autopsy on a dead patient.

6. So far we have been using Pharo's System Browser to edit code. Seaside provides an alternate method of editing code—using the web browser. To do this we first need to turn on some Seaside tools using the 'Halos' link at the bottom of the page. While we could do this in the current web browser's window, it will be helpful for what follows to keep the existing page and have a second page or tab open on the tools. How you do this will depend on your web browser, but most modern browsers support opening a link in a new tab (generally with a right-click context-sensitive menu). Following shows the result of opening 'Halos' in a new tab (click 'Halos' again if the box does not appear).



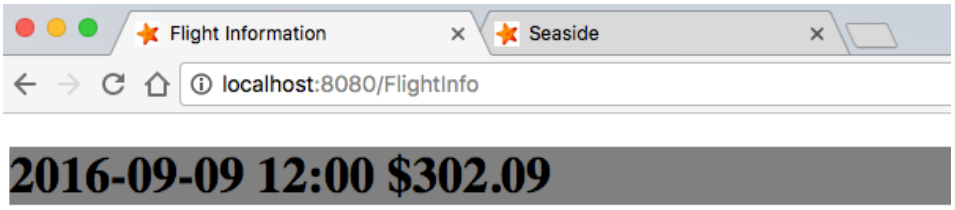
What Seaside has done here is add a box around our component that provides some information and tools (this is the 'halo'). The halo shows the component class name (FlightInfoComponent), three icons (that bring up a Class Browser, an Object Editor, and a CSS Style Editor), and links for Render and for Source). We will examine each of these.
Note that the 'Render' at the top right is bold. This lets us know that Seaside is 'rendering' the generated HTML. Click on the 'Source' to show the Source HTML. We can see that the 'heading:' message causes Seaside to render the model inside an <h1> element.



Click on the 'Render' to return to 'render' mode.

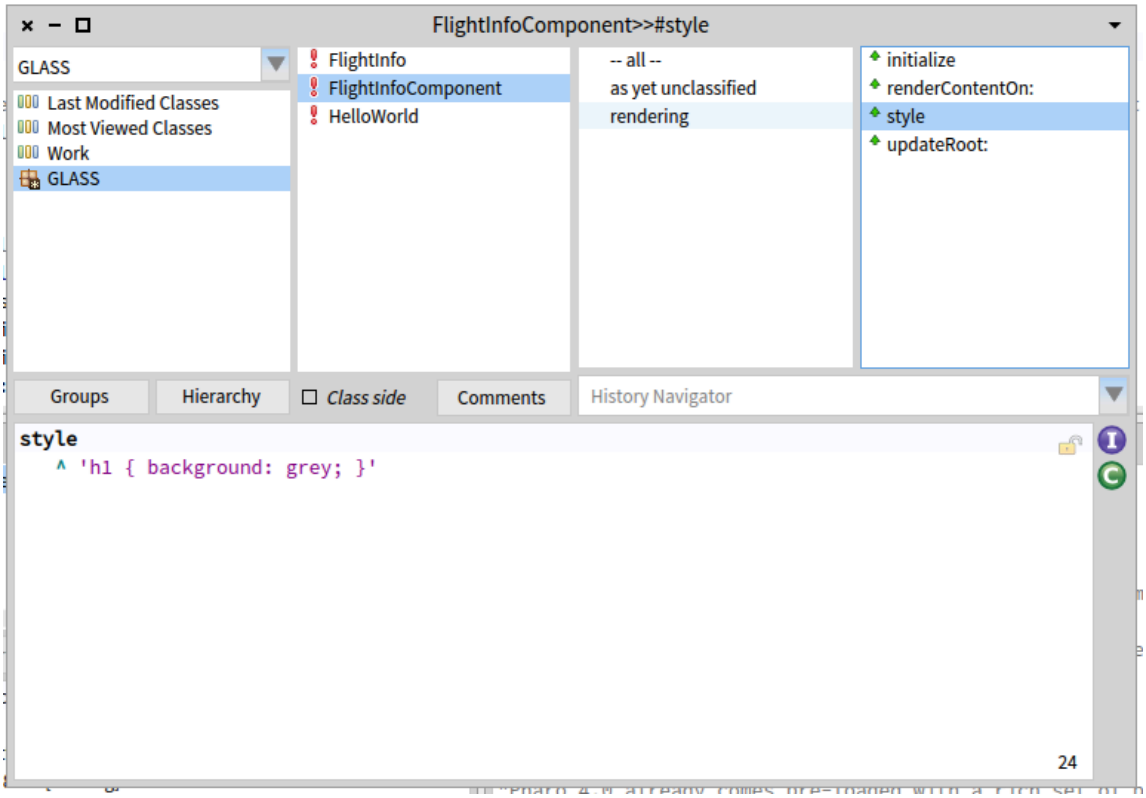
7. Next, click on the third icon that shows a tool over three colored circles. This changes the page to a CSS Style editor. Enter 'h1 { background: grey; }' in the text area and click the 'Save' button at the bottom (not shown here).

8. Now switch to the page (tab or window) with the original application and refresh. You should see a new background on the <h1> element.



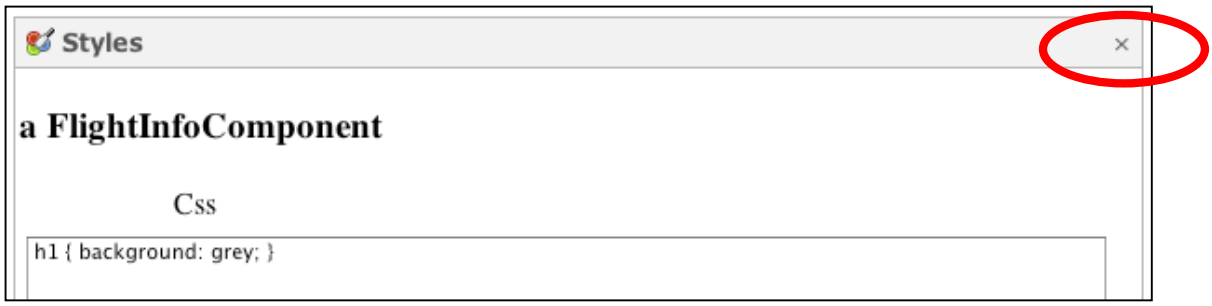
Depending on your environment, the grey might not show up well. If you don't see any change, go back to step #7 and try 'red' or 'yellow'.

9. Return to Pharo and click on 'FlightInfoComponent' in the System Browser, then on 'style' in the method list. You can see that Seaside has added a method to your component that provides CSS information.



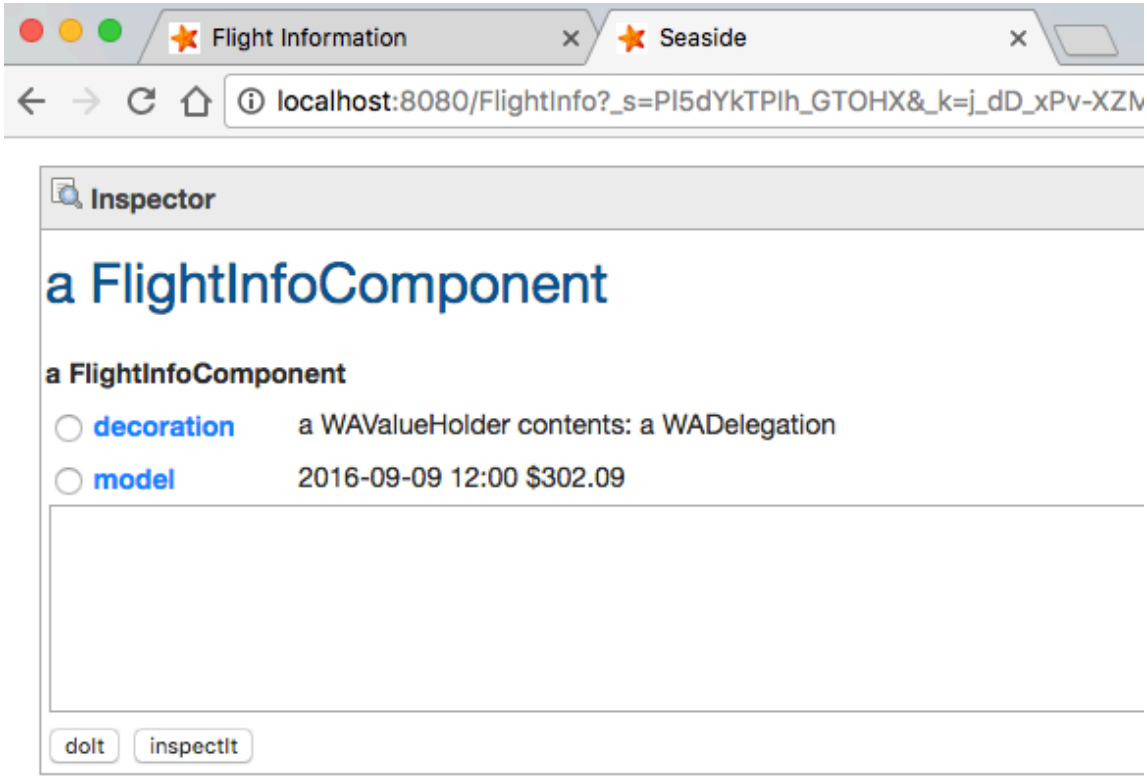
10. To reduce future confusion, edit this method so that the background for h1 is 'white'.

11. The next tool to explore is the Object Inspector. Return to the CSS Style Editor page in your web browser, and click the 'X' in the top right of the page.

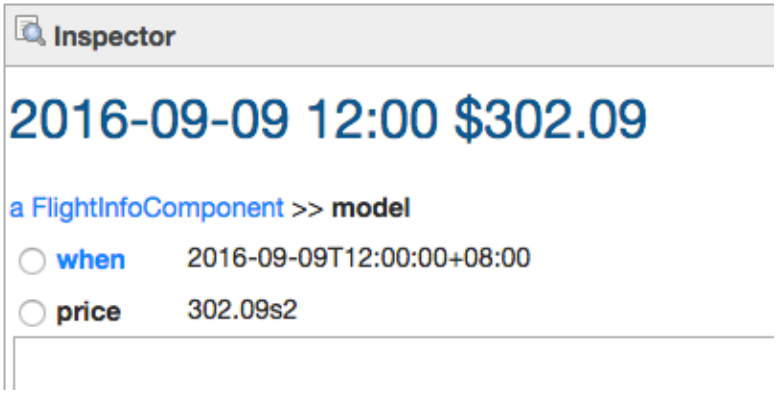


This should return you to the page shown above in step #6.

12. In the halos, the middle icon is a magnifying glass over a page. Click on that icon to open an Object Inspector.



13. In this Inspector you are first shown an inspector on the instance of FlightInfoComponent being presented by Seaside. Recall that we defined the class with one instance variable, 'model.' What we didn't notice is that one of our superclasses defined another instance variable, 'decoration.' Click on the 'model' link to inspect the instance of FlightInfo and see its instance variables.



14. In web browser, in the text area below price, type the following and click the 'do it' button.

```
self addHours: 2.
```

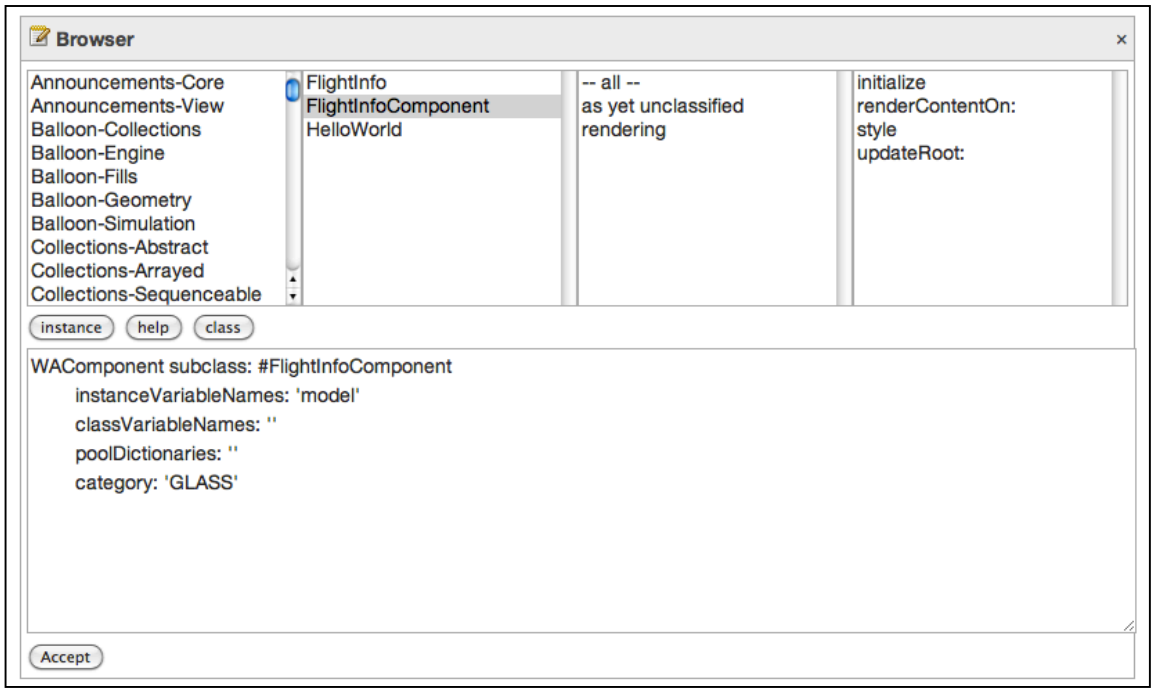
Notice how this changes the time (and the price, since the 'addHours:' method calls 'calculatePrice').

15. Again, in the text area, type the following and click on the 'inspect it' button. This will show you an inspector on 4.0, an instance of Float. From this we can see that we essentially have a Workspace available through the web.

```
16 sqrt.
```

When you are done inspecting objects, click the 'X' in the top right to close the Object Inspector. This should return you to the halos shown in step #6.

16. The final tool available from the halos is a Class Browser. Click on the first icon showing a spiral bound notepad with a pencil over it. This will open a Class Browser that should look a lot like the System Browser in Pharo. ~~We can use this browser (as an alternative to Pharo's System Browser) to add a few methods to our application component.~~



17. At this point our application simply initializes a model and displays it. By itself, this isn't a very sophisticated application and doesn't demonstrate much HTML functionality. The simplest addition that actually interacts with the data is a link. We will modify the render method to add an anchor, give it some code to execute when the link is clicked, and give it some text to display. In ~~the web browser~~ *Pharo*, click on 'renderContentOn:' in the last column and edit the text to match the following and click the 'Accept' button.

```
renderContentOn: html

   html heading: model.
   html anchor
      callback: [model addHours: -30];
      with: '<- Earlier'.
```


The added lines send one unary message ('anchor') and two keyword messages ('callback:' and 'with:'). As discussed in more detail below (at step #18), the 'addHours:' message will not be sent when this method is called. We have seen a message cascade before (in chapter 3), so we know that the receiver of the 'with:' message is the same as the receiver of the 'callback:' message. So which object is the receiver of the 'callback:' message? Since unary messages take precedence over keyword messages, the 'anchor' message will be sent to the passed-in html, and an object will be returned. The object is an instance of WAAnchorTag, a Seaside class that represents the <a> element in an HTML document.

The use of the cascade syntax here is important. We want to create one anchor and send it two messages. If we had written the code without the cascade, we might have been tempted to do the following:

```
html heading: model.
html anchor callback: [model addHours: -30].
html anchor with: '<- Earlier'.  "WRONG!"
```

This code would have created two anchors and sent the 'callback:' message to one and the 'with:' message to the other. This is not what we want! The correct way to do this (without using the cascade) would be to use a temporary variable, but it is more complex than the cascade approach. This, then, is an example of where the cascade makes the code simpler and more readable.

```
| myAnchor |
html heading: model.
myAnchor := html anchor.
myAnchor callback: [model addHours: -30].
myAnchor with: '<- Earlier'.
```

Now, let's discuss the two keyword messages sent to the new anchor. The simpler one comes last. The 'with:' message is being sent to the anchor tag with a single argument. The argument is a string literal, identified in Smalltalk with a straight single-quote character at the beginning and end. (If you need to insert a one single quote inside a string literal, put two in.) In Seaside, the 'with:' message should be the last one sent to a tag since it causes the content to actually be written to the HTML page. (Since it must be last, I don't use the 'yourself' at the end.)

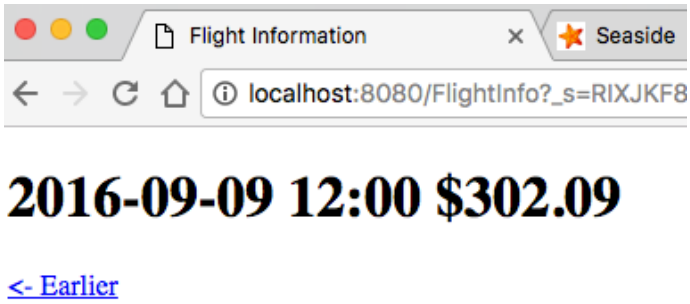
18. The 'callback:' keyword message introduces a new Smalltalk concept—a code block. In Smalltalk, any code inside square brackets ([]) is code that is *simply an object* that can be passed as an argument to any method that accepts an argument. Like any other object, it can be referenced by variables and messages can be sent to it at some later time. Thus, in our 'renderContentOn:' method we are not actually sending the 'addHours:' message; instead we are defining a block of code that *might* be executed later, at which time the 'addHours:' message will be sent.

A code block object knows its context—that is, the method in which it was defined. It also knows the method arguments and temporaries that existed at the time it was created. When the expressions contained in the block are later evaluated (by sending the message 'value' to the block), the expressions will properly reference instance variables, method arguments, and method temporaries.

By sending a code block to an anchor tag as the argument to a 'callback:' message, we are telling the anchor tag to hold onto this block, and when a user clicks on the anchor on the web page, Seaside will send the 'value' message to the code block, which will send the 'addHours:' message to the page's model with the argument of negative 30.

These Anonymous Functions (see http://en.wikipedia.org/wiki/Anonymous_function) give Smalltalk much of its power and flexibility. You might be familiar with some related approaches in other languages (JavaScript, Perl, Ruby, and of course Lisp have similar concepts). In C you can pass a pointer to a function as an argument to a function and in Java you can define inner classes, though these don't have the full capabilities of Smalltalk's blocks.

19. After saving the changes to the 'renderContentOn:' method, go back to your web browser and refresh the page. You should see a link below the heading and clicking on the link should cause the information to change.



20. Next we will add a link to move to a later time. This time rather than doing the call directly in the block, we will call a local method that updates the model. Add the method 'later' to FlightInfoComponent, and then modify the 'renderContentOn:' method to call this new method. (These edits can be made from Pharo or from a web browser.) In this case we still have a code block being passed to the new anchor tag, but the code block calls a local method that sends a message to the model.

    Which approach you take is a matter of style and will generally be driven by the complexity of the callback operation. If you have any more that one message send, then the code should probably be put in its own method. A code block can be many lines long, but that would only clutter the 'renderContentOn:' method, which is already several lines long! In Smalltalk it is considered poor form to have a method longer than you can read in the code browser without scrolling, or about 6-8 lines long.

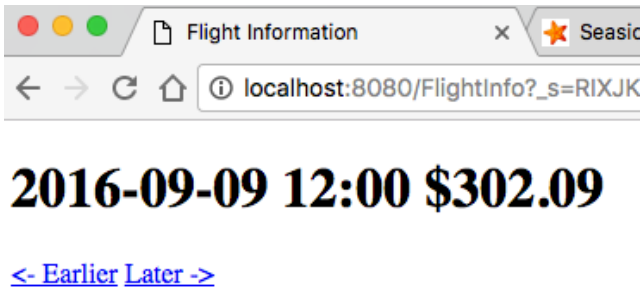
```
later

   model addHours: 30.
```

```
renderContentOn: html

   html heading: model.
   html anchor
      callback: [model addHours: -30];
      with: '<- Earlier'.
   html space.
   html anchor
      callback: [self later];
      with: 'Later ->'.
```

21. Try out the new page in your web browser. Click the links and note how the data changes.



22. Save your Pharo image before we go on to explore some other aspects of Seaside.

