Chapter 1

Seaside by Example

Alex ►*A short intro before the first section will make this chapter homogenous with other chapters.*◄

Alex ►*We should also mention that Seaside embeds a web server. This is not clear.*◄

1.1 What is Seaside?

Seaside is a web application framework for Smalltalk originally developed by Avi Bryant in 2002. A couple of the better known applications of Seaside are SqueakSource[1] and Dabble DB[2]. Seaside is unusual in that it is thoroughly object-oriented — there are no XHTML templates, no complicated control flows through web pages, and no encoding of state in URLs — you just send messages to objects.

Modern web application development frameworks have to cope with a host of problems. Expressing non-trivial control flows across multiple web pages is often cumbersome. Many web applications forbid the use of the browser's "back" button due to the difficulty of keeping track of the state of a session. Multiple control flows can also be difficult or impossible to express.

Seaside is a component-based framework that uses "continuations"[3] to keep track of multiple points in the control flow of web applications. Continuations are managed automatically by Seaside, so web developers

[1] http://SqueakSource.com

[2] http://DabbleDB.com

[3] A *continuation* represents "the rest of the computation" at any point in a computation. In Smalltalk, a continuation is just an object that captures the current state of the computation, and which can be resumed at any point.

do not even have to be aware of the underlying machinery. It just works.

Seaside makes web development easier in the following ways: Control flow can be expressed naturally in terms of messages sends. Seaside keeps track of the state of each user session. The browser's "back" button will work — Seaside keeps track for you which web page corresponds to which point in the execution of the web application. Backtracking of state can be enabled, so that navigation back in time will undo side-effects. Alternatively, transaction support is available too, to prevent users from undoing permanent side-effects using the back button. The developer does not have to encode any information in the URL — this too is managed automatically for you.

Web pages are built up from nested components, each of which may supports its own, independent control flow. There are no XHTML templates — instead valid XHTML is generated programmatically using a simple Smalltalk API. Seaside supports Cascading Style Sheets (CSS), so content and layout are cleanly separated.

Finally, Seaside provides a convenient web-based development interface, making it easy to develop applications iteratively, debug applications interactively, and recompile and extend applications while the server is running.

1.2 Getting started

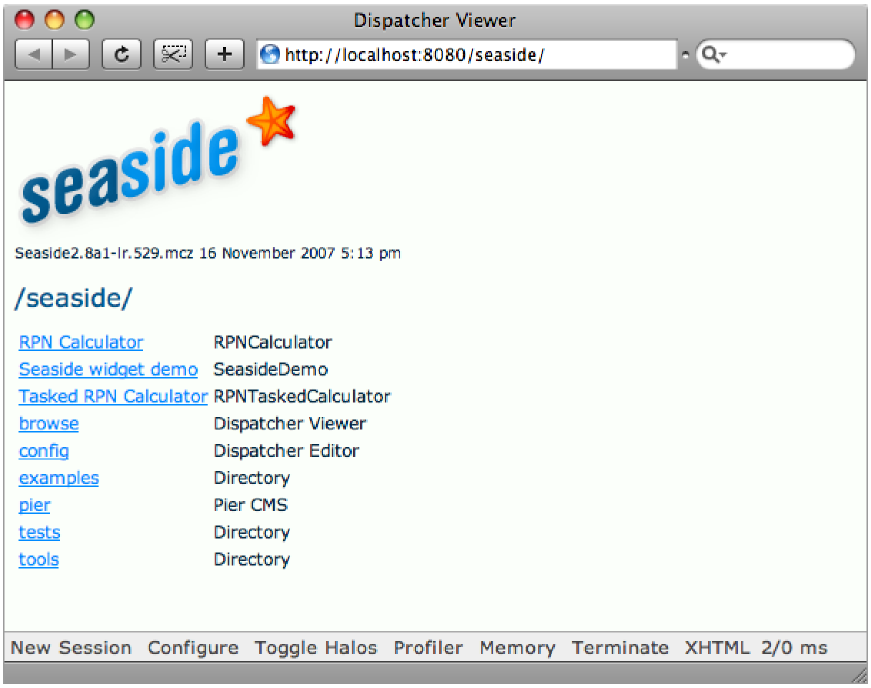
The easiest way to get started is to download the "Seaside One-Click Experience" from the Seaside web site[4]. This is a prepackaged version of Seaside 2.8 for Mac OSX, Linux and Windows. The same web site lists many pointers to additional resources, including documentation and tutorials. Be warned, however, that Seaside has evolved considerably over the years, and not all available material refers to the latest version of Seaside.

An alternative to the "one-click" image is to download the latest Squeak Developers' Web image[5], which comes with many useful tools pre-loaded. If you are feeling more adventurous, you can install Seaside yourself into an image by following the manual installation instructions on the Seaside web site.

You can turn the Seaside server on and off respectively by evaluating `WAKom startOn: 8080` or `WAKom stop`. The default administrator login and password in the prepackaged installation is `admin/seaside`. To change this, simply evaluate: `WADispatcherEditor initialize`

[4] http://seaside.st

[5] http://damien.cassou.free.fr/squeak-dev.html



Dispatcher Viewer

http://localhost:8080/seaside/

Seaside2.8a1-lr.529.mcz 16 November 2007 5:13 pm

/seaside/

RPN Calculator	RPNCalculator
Seaside widget demo	SeasideDemo
Tasked RPN Calculator	RPNTaskedCalculator
browse	Dispatcher Viewer
config	Dispatcher Editor
examples	Directory
pier	Pier CMS
tests	Directory
tools	Directory

New Session Configure Toggle Halos Profiler Memory Terminate XHTML 2/0 ms

Figure 1.1: Starting up Seaside

Start the Seaside server and open a browser to http://localhost:8080/seaside/*. What you see should look like Figure 1.1. Now go to* examples ▷ counter*. This is a simple example of a counter that can be incremented or decremented by clicking on the ++ and –– links. Play with the counter by clicking on these links. Use your browser's "back" button to go back to a previous state, and then click on ++ again. Notice how the counter is correctly incremented with respect to the current displayed state, rather than the state the counter was left in when you started using the "back" button.*

Now notice the toolbar at the bottom of the web page. New Session will start a new session on the counter application. Configure allows you to configure the settings of your application through a convenient web-interface. (To close the Configure view, click on the x in the top right corner.) Toggle Halos provides you with an interface to explore the state of the running application. Profiler and Memory provide you with detailed information about the run-time performance of the application. XHTML can be used to validate the generated web page. (This only works when the application is publicly accessible from the web, as it uses the W3C validation service.)

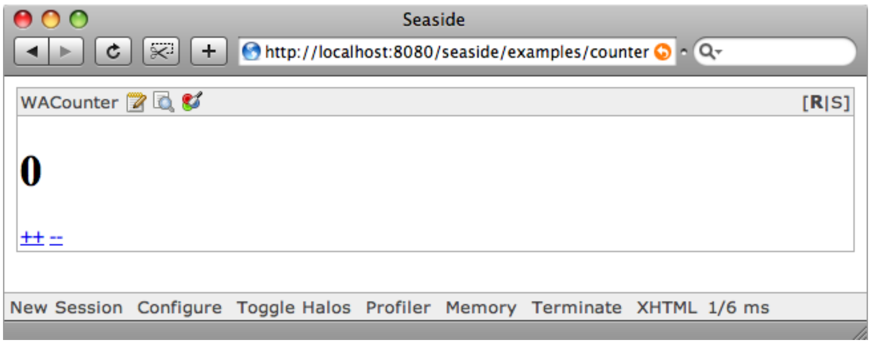
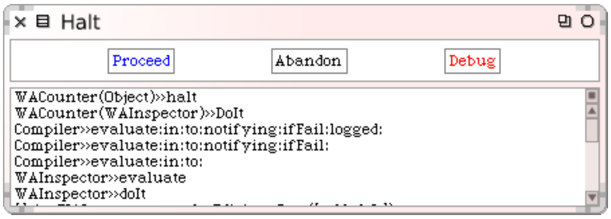

Seaside
http://localhost:8080/seaside/examples/counter
WACounter
[R|S]
0
++ --
New Session Configure Toggle Halos Profiler Memory Terminate XHTML 1/6 ms

Figure 1.2: Halos

Select Toggle Halos*. (See Figure 1.2.) At the top left you will see the text* WACounter*. This is the class of the Seaside component that implements the behavior of this web page. Next to this are three icons. The first activates a Seaside class browser opened to this class. The second opens an object inspector on the currently active instance. The third displays the CSS style sheet for this component. At the top right you can toggle between the rendered view and the source view of the web page. Experiment with all of these buttons. Note that the source view contains active links. Contrast the source view provided by the Halos with the source view offered by your browser.*

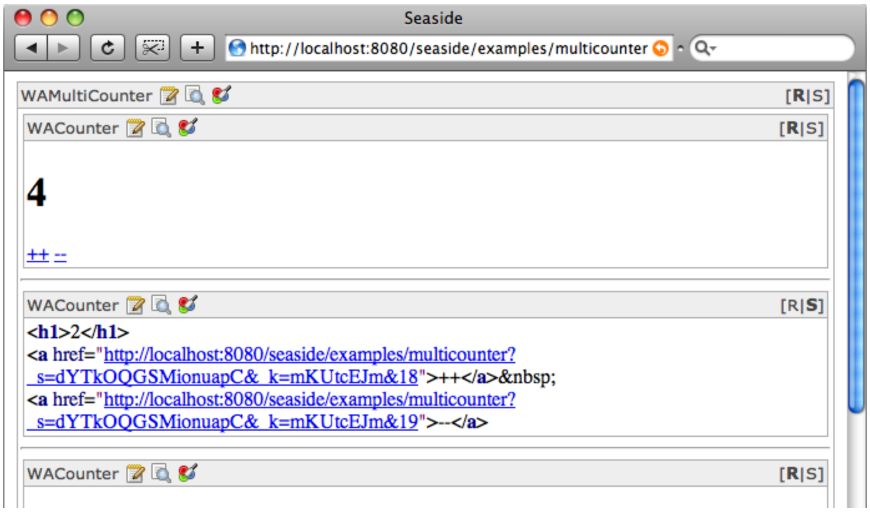
The class browser and object inspector available from Seaside are provided for convenience. For development purposes you will normally be working from the currently running image.



```
WACounter(Object)>>halt
WACounter(WAInspector)>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:logged:
Compiler>>evaluate:in:to:notifying:ifFail:
Compiler>>evaluate:in:to:
WAInspector>>evaluate
WAInspector>>doIt
```

Figure 1.3: Halting the counter

Open an inspector on the counter within the browser and evaluate self halt*. Notice that the web page will stop loading. Now switch to the Seaside image. You* *should now see a pre-debugger window (Figure 1.3) open on the current instance. Browse this instance in the debugger, and then* Proceed*. Go back to the web browser and note that the counter application is running again.*



```
<h1>2</h1>
<a href="http://localhost:8080/seaside/examples/multicounter?
_s=dYTkOQGSMionuapC&_k=mKUtcEJm&18">++</a>&nbsp;
<a href="http://localhost:8080/seaside/examples/multicounter?
_s=dYTkOQGSMionuapC&_k=mKUtcEJm&19">--</a>
```

Figure 1.4: Independent subcomponents

Open http://localhost:8080/seaside/examples/multicounter*. You will see an application built out of a number of independent instances of the counter component. Increment and decrement several of the counters. Verify that they behave correctly even if you use the "back" button. Toggle the halos to see how the application is built out of nested components. Use the Seaside class browser to view the implementation. (There should be three methods each on the class and instance sides.)*

Point your browser to http://localhost:8080/seaside/config*. Supply the login and password* admin/seaside *(or whatever you defined them to be). Note that you can configure, copy or remove individual components. Select* Configure *next to "examples". Add a new entry point called "counter2". Set the root component to* WACounter*, then* Save *and* Close*. Now we have a new counter installed at* http://localhost:8080/seaside/examples/counter2*. Use the same configuration interface to remove this entry point.*

The toolbar is only available during development. You can either use the configuration interface or the Configure button in the toolbar to set the deployment mode from false to true. Note that this only affects new sessions. You can also set the deployment mode on or off globally by evaluating WAGlobalConfiguration setDeploymentMode or WAGlobalConfiguration setDevelopmentMode.

If you remove the config application, you can get it back by evaluating: WADispatcherEditor initialize

1.3 Seaside components

Let's have a closer look at how Seaside works.

Seaside applications are built out of components. Every Seaside component should inherit directly or indirectly from WAComponent. (See Figure 1.6.)

Define a subclass of WAComponent *called* WAHelloWorld.

Components must know how to render themselves. Usually this is done by implementing the method renderContentOn:, which gets as its argument an instance of WAHtmlCanvas that knows how to render XHTML.

Implement the following method, and put it in a method category called rendering*:*

```
WAHelloWorld»renderContentOn: html
   html text: 'hello world'
```

Now we must inform Seaside that this component is allowed to be a standalone application. **Alex** ►*is this "allow to" correct?*◄

Implement the following method on the class side of WAHelloWorld.

```
WAHelloWorld class»canBeRoot
   ↑ true
```

You are almost done now.

Go to http://localhost:8080/seaside/config*, add a new entry point called "hello", and set its root component to be* WAHelloWorld*. Now point your browser to* http://localhost:8080/seaside/hello*. That's it!*

Simple and nested components

The counter and multi-counter examples are only slightly more complex than the "hello world" application.

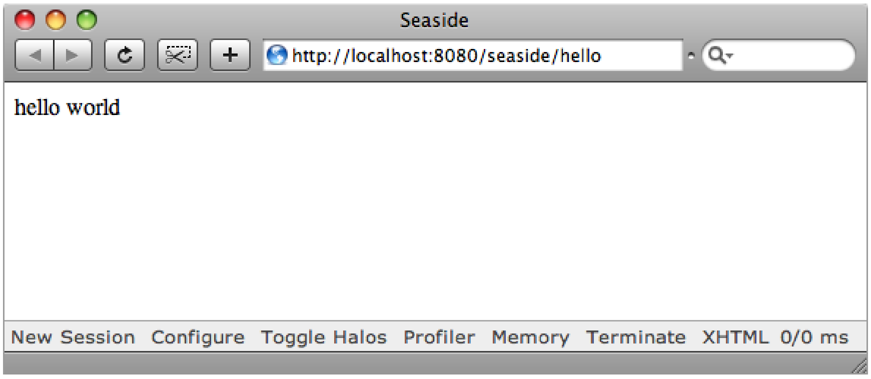

Seaside
http://localhost:8080/seaside/hello
hello world
New Session Configure Toggle Halos Profiler Memory Terminate XHTML 0/0 ms

Figure 1.5: "Hello World" in Seaside

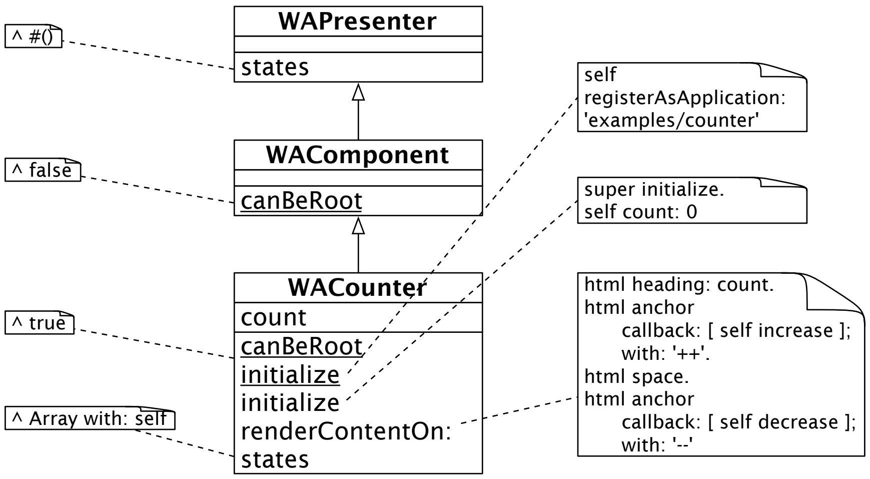


Figure 1.6: WACounter

The class WACounter is a standalone application, so it implements canBeRoot on the class side. It also automatically registers itself as an application in its class-side initialize method (see Figure 1.6).

Alex ▶*increase and decrease are not shown in Figure 1.6*◀ WACounter defines an instance variable count to keep track of the state of the counter. Since we want Seaside to synchronize the state with the browser page, *i.e.*, in case the user clicks on the browser's "back" button, we must inform Seaside which variables to track by implementing the states method. This should return an array of all objects to be tracked. In this case WACounter asks to track its own state by returning Array with: self.

Caveat. There is a subtle but important point to watch for when declaring state for backtracking. Seaside tracks state by making a *snapshot* of all the objects declared in the states array. WASnapshot is a subclass of IdentityDictionary that registers each object to be tracked as a key, and a (shallow) copy of its state as a value. If the state is restored from a snapshot, all registered objects are restored to the saved value. In the case of the counter, the state to be tracked is a number, which is *replaced* every time the count is incremented or decremented. The numbers themselves don't change, since they are immutable! This means that WACounter»states must return Array with: self and not Array with: count.

The rendering of the counter is relatively straightforward. The current value of the counter is displayed as an XHTML heading, and the increment and decrement operations are implemented as html anchors with callbacks to methods increase and decrease (not shown).

We will have a closer look at the rendering protocol in a moment. First let us have a quick look at the multi-counter.

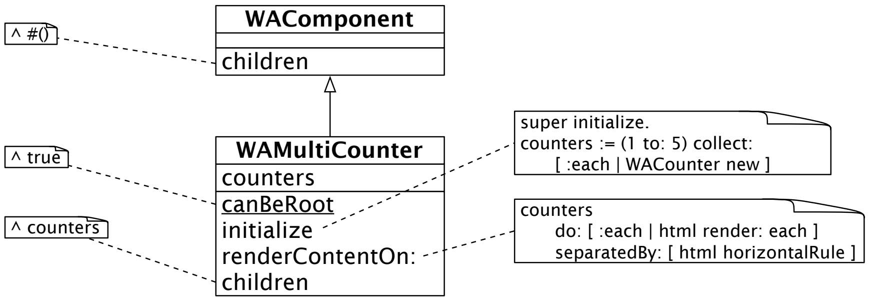


Figure 1.7: WAMultiCounter

It is also a standalone application, so it implements canBeRoot. In addition, it is a composite component, so Seaside requires it to declare its children by implementing a method children, which returns an array of all the components it contains. It trivially renders itself by rendering each of its subcomponents, separated by a horizontal rule. Aside from instance and class-side initialization methods, there is nothing else to the multi-counter!

Rendering XHTML

Seaside does not use templates to generate web pages. Instead XTHML is generated programmatically. We have already seen a couple of examples of how this is done.

Basically a Seaside component should implement the method renderContentOn:, which will be called by the framework whenever the component needs to be rendered. This method will be passed an argument, by convention called html, which is a *canvas* for rendering the component.

Here are some of the most basic rendering methods:

```
html text: 'hello world'.  "render a plain text string"
html html: '&ndash;'.      "render an XHTML incantation"
html render: 1.            "render any object"
```

The message render: can be sent to a canvas to render any object (using double dispatch), though it is normally used to render subcomponents. This was the case in our multi-counter (see Figure 1.7).

A canvas provides a number of *brushes* that can be used to render content on the canvas. There are brushes for every kind of XHTML element such as paragraphs, tables, lists and so on. To see the full protocol of brushes and convenience methods available, you should browse the class WACanvas and its subclasses. html is actually an instance of the subclass WARenderCanvas.

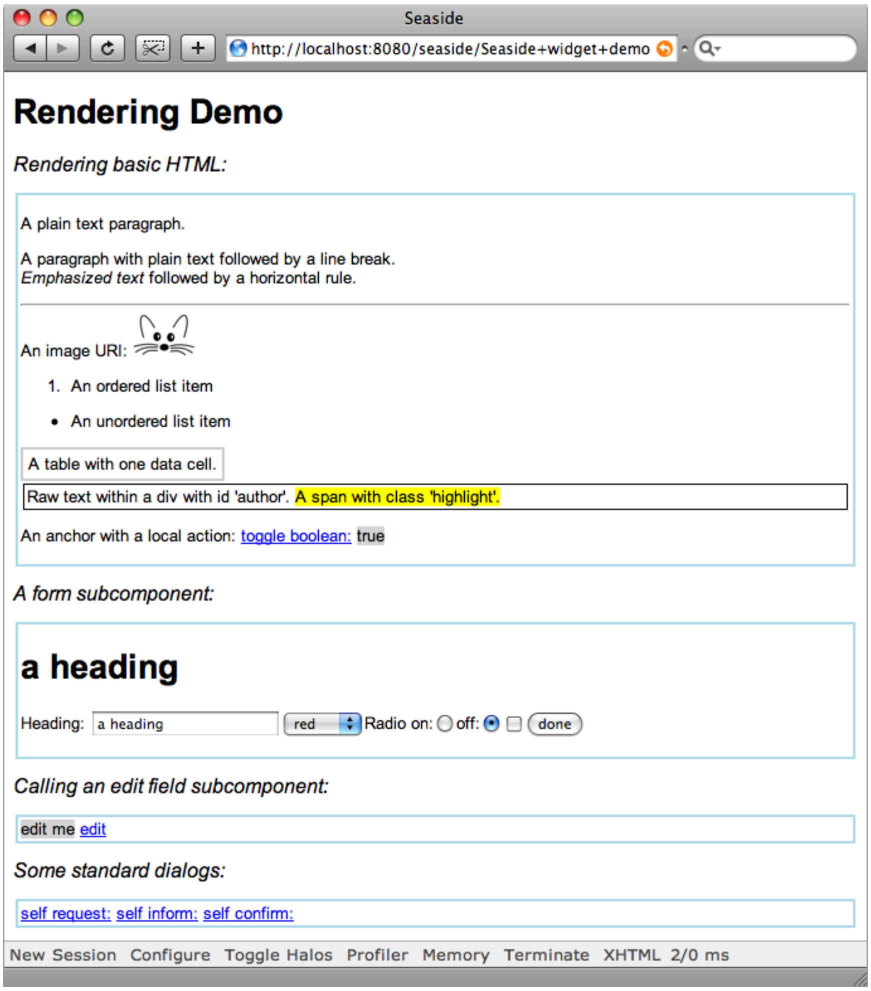
We have already seen the following brush used in the counter and multi-counter examples:

```
html horizontalRule.
```

In Figure 1.8 we can see a demo of many of the basic brushes offered by Seaside.[6] The root component SeasideDemo simply renders its subcomponents, which are instances of SeasideHtmlDemo, SeasideFormDemo, SeasideEditCallDemo and SeasideDialogDemo.

```
SeasideDemo»renderContentOn: html
  html heading: 'Rendering Demo'.
  html heading
    level: 2;
    with: 'Rendering basic XHTML: '.
  html div
    class: 'subcomponent';
    with: htmlDemo.
  "render the remaining components ..."
```

[6]See package SBE–SeasideDemo in the project http://www.squeaksource.com/SqueakByExample.



Rendering Demo

Rendering basic HTML:

A plain text paragraph.

A paragraph with plain text followed by a line break.
Emphasized text followed by a horizontal rule.

An image URI:

1. An ordered list item

- An unordered list item

A table with one data cell.

Raw text within a div with id 'author'. A span with class 'highlight'.

An anchor with a local action: toggle boolean: true

A form subcomponent:

a heading

Heading: a heading red Radio on: ○ off: ● ☐ done

Calling an edit field subcomponent:

edit me edit

Some standard dialogs:

self request: self inform: self confirm:

New Session Configure Toggle Halos Profiler Memory Terminate XHTML 2/0 ms

Figure 1.8: RenderingDemo

Recall that a root component must always declare its children, or Seaside will refuse to render them.

```
SeasideDemo»children
  ↑ Array with: htmlDemo with: formDemo with: editDemo with: dialogDemo
```

Note the two different ways of instantiating the heading brush. In the first case we set the text directly by sending the message heading:. In the second case, we instantiate the brush by sending heading, and then we send a cascade of messages to the brush to set its properties and render it. This is typical for many of the available brushes.

> If you send a cascade of messages to a brush including the message with:, then this should be the *last* message sent. This sets the content and renders the result.

The default heading is at level 1. We explicitly set the level of the second heading to 2. The subcomponent is rendered as an XHTML *DIV* with the CSS class "subcomponent". (More on CSS in Section 1.3.) Note as well that the argument to the with: keyword message need not be a literal string, but may be another component, or even — as in the next example — a block containing further rendering actions.

The SeasideHtmlDemo component demonstrates many of the most basic brushes. Most of the code should be self-explanatory.

```
SeasideHtmlDemo»renderContentOn: html
   self renderParagraphsOn: html.
   self renderListsAndTablesOn: html.
   self renderDivsAndSpansOn: html.
   self renderLinkWithCallbackOn: html
```

It is common practice to break up long rendering methods into many helper methods, as we have done here.

> Don't put all your rendering code into a single method. Split it into helper methods named after the pattern render*On:. All rendering methods go in method category *rendering*. Don't send renderContentOn: from your own code, use render: instead.

The first helper method shows how to generate XHTML paragraphs, plain and emphasized text, and images. Note that simple elements are rendered by directly specifying the text they contain, whereas as complex elements are specified using blocks.

```
SeasideHtmlDemo»renderParagraphsOn: html
   html paragraph: 'A plain text paragraph.'.
   html paragraph: [
      html
         text: 'A paragraph with plain text followed by a line break. ';
         break;
         emphasis: 'Emphasized text ';
         text: 'followed by a horizontal rule.';
         horizontalRule;
         text: 'An image URI: '.
      html image
         url: self squeakImageUrl;
         width: '50']
```

The next helper method shows how to generate lists and tables. A table even uses two levels of blocks to display each of its rows and the cells within the rows.

```
SeasideHtmlDemo»renderListsAndTablesOn: html
   html orderedList: [
      html listItem: 'An ordered list item'].
   html unorderedList: [
      html listItem: 'An unordered list item'].
   html table: [
      html tableRow: [
         html tableData: 'A table with one data cell.']]
```

The next example shows how we can specify CSS DIVs and SPANs with class or id attributes. Of course, the messages class: and id: can also be sent to the other brushes, not just to DIVs and SPANs. The method SeasideDemoWidget»style defines the display attributes for XHTML elements with these attributes (see Section 1.3).

```
SeasideHtmlDemo»renderDivsAndSpansOn: html
   html div
      id: 'author';
      with: [
         html text: 'Raw text within a div with id ''author''. '.
         html span
            class: 'highlight';
            with: 'A span with class ''highlight''.']
```

Finally we see a simple example of a link, created by binding a simple callback to an anchor. Clicking on the link will cause the subsequent text to toggle between "true" and "false".

```
SeasideHtmlDemo»renderLinkWithCallbackOn: html
   html paragraph: [
      html text: 'An anchor with a local action: '.
      html span with: [
         html anchor
            callback: [toggleValue := toggleValue not];
            with: 'toggle boolean:'].
```

```
    html space.
    html span
      class: 'boolean';
      with: toggleValue ]
```

Alex ►*I would stress that toggleValue has to be a class attribute. It cannot be a temporary variable (method or block)*◄

> Note that actions should only appear in callbacks. You should not change the state of the application while you are rendering it.

Forms

Forms are rendered just like the other examples we have seen till now. Here is the code for the SeasideFormDemo component in Figure 1.8.

```
SeasideFormDemo»renderContentOn: html
  | radioGroup |
  html heading: heading.
  html form: [
    html span: 'Heading: '.
    html textInput on: #heading of: self.
    html select
      list: self colors;
      on: #color of: self.
    radioGroup := html radioGroup.
    html text: 'Radio on:'.
    radioGroup radioButton
      selected: radioOn;
      callback: [radioOn := true].
    html text: 'off:'.
    radioGroup radioButton
      selected: radioOn not;
      callback: [radioOn := false].
    html checkbox on: #checked of: self.
    html submitButton
      text: 'done' ]
```

Since a form is a complex entity, it is rendered using a block. Alex ►*I would love to see a better justification for using blocks. It is not that clear*◄ Note that state changes are all specified as callbacks.

There is one Seaside trick here that is worth special mention, namely the message on:of:, which is used to bind the text input field to accessors for the variable heading. Anchors and buttons also support this protocol. The first argument is the name of an instance variable for which accessors have been defined Alex ▶*Which accessors ? Shall I define heading and heading:, accessor and mutator of the variable heading? Does these accessors have to follow a naming convention ?*◀ , and the second argument is the object to which this instance variable belongs. In the case here of a text input field, this saves us the trouble if having to define a callback which updates the field as well as having to bind the default contents of the field to the current value. The heading variable is automatically updated whenever we update the text field.

The same trick is used twice more in this example, to cause the selection of a color to automatically update the color variable, and to bind the result of checking the checkbox to value of the checked variable.

Many further examples are available directly in the functional tests for Seaside. Have a look at the system category *Seaside-Tests-Functional*, or just point your browser to http://localhost:8080/seaside/tests/alltests. The Form Elements example illustrates most of the features of forms.

Don't forget, if you Toggle Halos, you can directly browse all the source code of the examples on line using the Seaside class browser.

Cascading style sheets

Cascading Style Sheets[7], or CSS for short, have emerged as a standard way for web applications to separate style from content. Seaside relies on CSS to avoid cluttering your rendering code with layout considerations.

You can set the CSS style sheet for your web components by defining the method style, which should return a string containing the CSS rules for that component. The styles of all the components displayed on a web page are joined together, so each component can have its own style. A better approach can be to define an abstract class for your web application that defines a common style for all its subclasses.

Actually, for deployed applications, it is more common to define style sheets as external files. This way the look and feel of the component is completely separate from its functionality. (Have a look at WAFileLibrary, which provides a way to serve static files without the need for a standalone server.)

If you already are familiar with CSS, then that's all you need to know. Otherwise, read on for a very brief introduction to CSS.

Basically a CSS style sheet consists of a set of rules that specify how to format given XHTML elements. Each rule consists of two parts. There is a

[7] http://www.w3.org/Style/CSS/

```
SeasideDemoWidget»style
   ↑ '
body {
   font: 10pt Arial, Helvetica, sans-serif, Times New Roman;
}
h2 {
   font-size: 12pt;
   font-weight: normal;
   font-style: italic;
}
table { border-collapse: collapse; }
td {
   border: 2px solid #CCCCCC;
   padding: 4px;
}
#author {
   border: 1px solid black;
   padding: 2px;
   margin: 2px;
}
.subcomponent {
   border: 2px solid lightblue;
   padding: 2px;
   margin: 2px;
}
.highlight { background-color: yellow; }
.boolean { background-color: lightgrey; }
.field { background-color: lightgrey; }
'
```

Figure 1.9: SeasideDemoWidget common style sheet.

selector that specifies which XHTML elements the rule applies to, and there is a *declaration* which sets a number of attributes for that element.

Figure 1.9 illustrates a particularly simply style sheet for the rendering demo. The first rule specifies a preference for the fonts to use for the *body* of the web page. The next few rules specify properties of second-level headings, tables, and table data.

The remaining rules have selectors which will match XHTML elements that have the given "id" or "class" attributes. The main difference between these two is that many elements may have the same class, but only one element may have a given id (*i.e.*, an *identifier* must identify a *unique* element on the page, such as a menu, last modified date, or author, whereas a class attribute, such as highlighted text, may occur multiple times on any page).

Note that a given XHTML element may have multiple classes, thus enabling a fine degree of control over how display attributes are combined.

This style sheet expects at most one element to specify the *author* of the web page. To match an id, the selector should include the name of the id preceded by a #. A class name should be preceded by a dot.

Selector conditions may be combined, so the selector div.subcomponent will only match an XHTML element if it is both a DIV *and* it has a class attribute equal to "subcomponent".

It is also possible to specify nested elements, though this is seldom necessary. For example, the selector "p span" will only match a SPAN within a paragraph.

There are numerous books and web sites to help you learn CSS. For a dramatic demonstration of the power of CSS, we recommend you to have a look at the CSS Zen Garden[8], which shows how the same content can be rendered in radically different ways simply by changing the CSS style sheet.

Managing control flow

Seaside makes it particularly easy to design web applications with non-trivial control flow. There are basically two mechanisms that you can use:

1. A component caller can be temporarily replaced by another component, callee, by sending caller call: callee. The caller is usually self, but could also be any other currently visible component. The callee returns control by sending answer:.

2. A *task* is a special kind of component that subclasses WATask (instead of WAComponent). Instead of defining renderContentOn:, it defines no content of its own, but rather defines a go method that sends a series of call: messages to activate various subcomponents in turn.

Call and answer

call: and answer: should never be used while rendering. They may safely be sent from within a callback, or from with the go method of a task.

There is a trivial example of call: and answer: in the rendering demo of Figure 1.8. The component SeasideEditCallDemo displays a text field and an *edit* link. The callback for the edit link calls a new instance of

[8] http://www.csszengarden.com/

SeasideEditAnswerDemo initialized to the value of the text field. The callback also updates this text field to the result which is sent as an answer.

```
SeasideEditCallDemo»renderContentOn: html
  html span
    class: 'field';
    with: self text.
  html space.
  html anchor
    callback: [self text: (self call: (SeasideEditAnswerDemo new text: self text))];
    with: 'edit'
```

What is particularly elegant is that the code makes absolutely no reference to a new web page that must be created. At run-time, only the SeasideEditCallDemo component will be replaced. The parent component and the other peer components are otherwise untouched by the call.

The SeasideEditAnswerDemo component is also remarkably simple. It just renders a form with a text field. The "submit" button is bound to a callback that will answer the final value of the text field.

```
SeasideEditAnswerDemo»renderContentOn: html
  html form: [
    html textInput
      on: #text of: self.
    html submitButton
      callback: [ self answer: self text ];
      text: 'ok'.
    ]
```

That's it.

Seaside takes care of the control flow and the correct rendering of all the components. Interestingly, the "back" button of the browser will also work just fine (though side effects are not rolled back unless we take additional steps).

Convenience methods

Since certain call/answer dialogues are very common, Seaside provides some convenience methods to save you the trouble of writing components like SeasideEditAnswerDemo. We can see the following convenience methods being used within SeasideDialogDemo»renderContentOn

The message request: performs a call to a component that will let you edit a text field. The component answers the edited string. (See Figure 1.10.) An optional label and default value may also be specified.

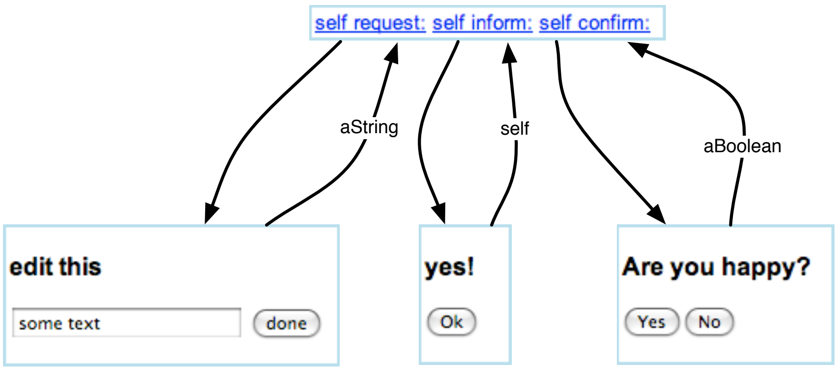


Figure 1.10: Some standard dialogs

```
html anchor
   callback: [ self request: 'edit this' label: 'done' default: 'some text' ];
   with: 'self request:'.
```

The message inform: calls a component that simply displays the argument message and waits for the user to click "ok". The called component just returns self.

```
html space.
html anchor
   callback: [ self inform: 'yes!' ];
   with: 'self inform:'.
```

The message confirm: asks a questions and waits for the user to select either "Yes" or "No". The component answers a boolean, which can be used to perform further actions.

```
html space.
html anchor
   callback: [
      (self confirm: 'Are you happy?')
         ifTrue: [ self inform: ':-)' ]
         ifFalse: [ self inform: ':-(' ]
      ];
   with: 'self confirm:'.
```

A few further convenience methods, such as chooseFrom:caption:, are defined in the *convenience* protocol of WAComponent.

Tasks

Alex ►◄

A task is a component that subclasses WATask. It does not render anything itself, but simply calls other components in a control flow defined by implementing the method go.

WAConvenienceTest is a simple example of a task defined in the system category *Seaside-Tests-Functional*. To see its effect, just point your browser to http://localhost:8080/seaside/tests/alltests and select Convenience.

```
WAConvenienceTest»go
  [ self chooseCheese.
    self confirmCheese ] whileFalse.
  self informCheese
```

This task calls in turn three components. The first is a WAChoiceDialog to choose a cheese (generated by the convenience method chooseFrom:caption:).

```
WAConvenienceTest»chooseCheese
  cheese := self
    chooseFrom: #('Greyerzer' 'Tilsiter' 'Sbrinz')
    caption: 'What''s your favorite Cheese?'.
  cheese isNil ifTrue: [ self chooseCheese ]
```

Alex ► *Is there a situation where cheese may be nil? Maybe if a browser authorize an empty selection...*◄

The second is a WAYesOrNoDialog to confirm the choice (generated by the convenience method confirm:). Finally a WAFormDialog is called (via the convenience method inform:).

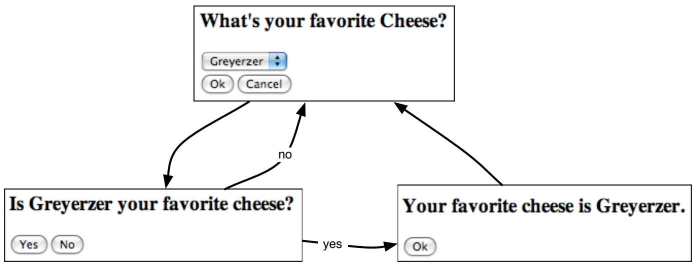

What's your favorite Cheese?
Greyerzer
Ok
Cancel
no
Is Greyerzer your favorite cheese?
Yes
No
yes
Your favorite cheese is Greyerzer.
Ok

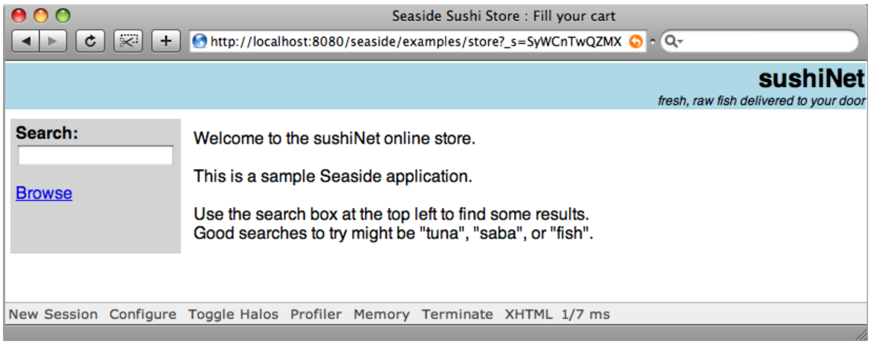
Figure 1.11: A simple task

Transactions

We have already seen that Seaside can keep track of the correspondence between the state of components and individual web pages by having components register their state for backtracking. It suffices to implement the method states and have it return an array of all the objects whose state must be tracked (*i.e.*, by having Seaside make shallow copies of these objects).

Sometimes, however, we do not want to track state, but rather *prevent* the user from accidentally undoing side effects that should be permanent. This is often referred to as "the shopping cart problem". Once you have checked out you shopping cart on a e-store and paid for the items you have purchased, it should not be possible to go back with the browser and add more items to the shopping cart!

Seaside allows you to prevent this by defining a task within which certain actions are grouped together as *transactions*. You can backtrack within a transaction, but once a transaction is complete, you can no longer go back to it. The corresponding pages are *invalidated*, and any attempt to go back to them will cause Seaside to generate a warning and redirect the user to the last valid page.


Seaside Sushi Store : Fill your cart
http://localhost:8080/seaside/examples/store?_s=SyWCnTwQZMX

sushiNet
fresh, raw fish delivered to your door

Search:
Browse

Welcome to the sushiNet online store.

This is a sample Seaside application.

Use the search box at the top left to find some results.
Good searches to try might be "tuna", "saba", or "fish".

New Session Configure Toggle Halos Profiler Memory Terminate XHTML 1/7 ms

Figure 1.12: The Sushi Store

The Seaside *Sushi Store* is sample application that illustrates many of the features of Seaside, including transactions. It should be bundled together with your installation of Seaside, in which case you can try it out by pointing your browser at http://localhost:8080/seaside/examples/store.[9]

The sushi store supports the following workflow:

[9]If you cannot find it in your image, there is a version of the sushi store available on SqueakSource from http://www.squeaksource.com/SeasideExamples/.

- Visit the store.
- Browse or search for sushi.
- Add sushi to your shopping cart.
- Checkout.
- Verify your order.
- Enter shipping address.
- Verify shipping address.
- Enter payment information.
- Your fish is on its way!

Alex ▶*An enumerate instead of an itemize would fit better I guess*◀

If you toggle the halos, you will see that the top-level component of the sushi store is an instance of WAStore. It does nothing but render the title bar, and then it renders an instance of WAStoreTask.

```
WAStore»renderContentOn: html
   "... render the title bar ..."
   html div id: 'body'; with: task
```

WAStoreTask actually capture the workflow above. At a couple of points it is critical that you not be able to go back and change the submitted information.

"Purchase" some sushi and then use the "back" button to try put more sushi into your cart. You will get the message "That page has expired."

Seaside allows you to express that certain part of a workflow act like a transaction — once the transaction is complete, you cannot go back. You can specify this very simply in a task by sending isolate: to the task with the transactional block as its argument. We can see this in the sushi store workflow as follows:

```
WAStoreTask»go
   | shipping billing creditCard |
   cart := WAStoreCart new.
   self isolate:
      [[self fillCart.
      self confirmContentsOfCart]
         whileFalse].
```

```
  self isolate:
     [shipping := self getShippingAddress.
     billing := (self useAsBillingAddress: shipping)
              ifFalse: [self getBillingAddress]
              ifTrue: [shipping].
     creditCard := self getPaymentInfo.
     self shipTo: shipping billTo: billing payWith: creditCard].

  self displayConfirmation.
```

Here we see quite clearly that there are two transactions. The first fills the cart and closes the shopping phase. (The helper methods fillCart etc. take care of instantiating and calling the right subcomponents.) Once you have confirmed the contents of the cart you cannot go back without starting a new session. The second transaction completes the shipping and payment data. You can navigate back and forth within the second transaction until you confirm payment. then both transactions are complete, and any attempt to navigate back will fail.

Transactions may also be nested.

A simple demonstration of this is found in the class WANestedTransaction. The argument block to the first isolate: send contains itself another, nested isolate: send:

Alex ▶*On the example below, I am wondering whether it makes sense to have a "self isolate: [self inform: 'Inside child txn']" since self inform: is a kind of atomic operation. There is no much to isolate here. why not to change the code with something like "self isolate: [self confirm: (self request: 'Inside child txn')]." I can do a back and forth within this isolate block in that case...*◀

```
WANestedTransaction»go
  self inform: 'Before parent txn'.
  self isolate:
        [self inform: 'Inside parent txn'.
        self isolate: [self inform: 'Inside child txn'].
        self inform: 'Outside child txn'].
  self inform: 'Outside parent txn'
```

Go to http://localhost:8080/seaside/tests/alltests and select Convenience *and select* Transaction*. Try to navigate back and forth within the parent and child transaction. Note that as soon as a transaction is complete, you can no longer go back inside the transaction without generating an error.*

1.4 A complete tutorial example

Let's see how we can build a complete Seaside application from scratch.[10] We will build a RPN (Reverse Polish Notation) calculator as a Seaside application that uses a simple stack machine as its underlying model. Furthermore, the Seaside interface will let us toggle between two displays — one which just shows us the current value on top of the stack, and the other which shows us the complete state of the stack. (See Figure 1.13.)

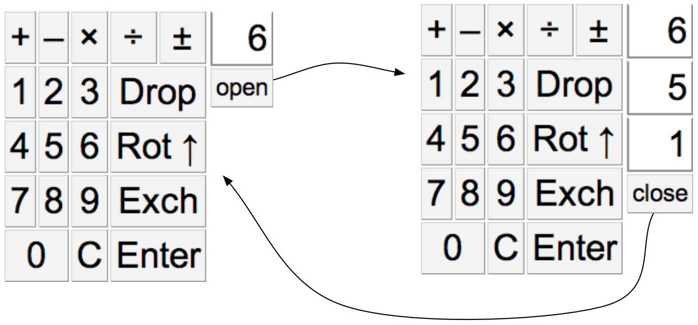


Figure 1.13: RPN calculator and its stack machine

We begin by implementing the stack machine and its tests.

Define a new class called MyStackMachine *with an instance variable* contents *initialized to a new* OrderedCollection.

```
MyStackMachine»initialize
   super initialize.
   contents := OrderedCollection new.
```

The stack machine should provide operations to push: and pop values, view the top of the stack, and perform various arithmetic operations to add, subtract, multiply and divide the top values on the stack.

Write some tests for the stack operations, and then implement these operations. Here is a sample test:

[10]The exercise should take at most a couple of hours. If you prefer to just look at the completed source code, you can grab it from the SqueakSource project http://www.squeaksource.com/SqueakByExample. The relevant category is *SBE-SeasideRPN*. The tutorial that follows uses slightly different class names so that you can compare your implementation with ours.

```
MyStackMachineTest»testDiv
   stack
      push: 3;
      push: 4;
      div.
   self assert: stack size = 1.
   self assert: stack top = (4/3).
```

You might consider using some helper methods for the arithmetic operations to check that there are two numbers on the stack before doing anything, and raising an error if this precondition is not fulfilled.[11] If you do this, most of your methods will just be one or two lines long.

You might also consider implementing MyStackMachine»printOn: to make it easier to debug your stack machine implementation with the help of an object inspector. (Hint: just delegate printing to the contents variable.)

Complete the MyStackMachine *by writing operations* dup *(push a duplicate of the top value onto the stack),* exch *(exchange the top two values), and* rotUp *(rotate the entire stack contents up — the top value will move to the bottom).*

Now we have a simple stack machine implementation. We can start to implement the Seaside RPN Calculator.

We will make use of 5 classes:

- MyWidget — this should be an abstract class that defines the common CSS style sheet for the application, and other common behavior for the components of the RPN calculator. It is a subclass of WAComponent and the direct superclass of the following three classes. **Alex** ►*four classes?*◄
- MyCalculator — this is the root component. It should register the application (on the class side), it should instantiate and render its subcomponents, and it should register any state for backtracking.
- MyKeypad — this displays the keys that we use to interact with the calculator.
- MyDisplay — this component displays the top of the stack and provides a button to call another component to display the detailed view.
- MyDisplayStack — this component shows the detailed view of the stack and provides a button to answer back. It is a subclass of MyDisplay.

[11] Better yet, use Object»assert: to specify the preconditions for each operation. This will raise an AssertionFailure if you try to use the stack machine in an invalid state.

Define MyWidget *in the category* MyCalculator*. Define the common* style *for the application.*

Here is a minimal CSS for the application. You can make it more fancy if you like.

```
MyWidget»style
   ↑ 'table.keypad { float: left; }
td.key {
   border: 1px solid grey;
   background: lightgrey;
   padding: 4px;
   text-align: center;
}
table.stack { float: left; }
td.stackcell {
   border: 2px solid white;
   border-left-color: grey;
   border-right-color: grey;
   border-bottom-color: grey;
   padding: 4px;
   text-align: right;
}
td.small { font-size: 8pt; }'
```

Define MyCalculator *to be a root component and register itself as an application (*i.e.*, implement* canBeRoot *and* initialize *on the class side). Implement* MyCalculator»renderContentOn: *to render something trivial (such as its name), and verify that the application runs in a browser.*

MyCalculator *is responsible for instantiating* MyStackMachine*,* MyKeypad *and* MyDisplay*. Define* MyKeypad *and* MyDisplay *as subclasses of* MyWidget*. All three components will need access to a common instance of the stack machine, so define the instance variable* stackMachine *and an initialization method* setMyStackMachine: *in the common parent,* MyWidget*. Add instance variables* keypad *and* display *to* MyCalculator *and initialize them in* MyCalculator»initialize*. (Don't forget to send* super initialize*!) Pass the shared instance of the stack machine to the keypad and the display in the same initialize method. Implement* MyCalculator»renderContentOn: *to simply render in turn the keypad and the display. To correctly display the subcomponents, you must implement* MyCalculator»children *to return an array with the keypad and the display. Implement placeholder rendering methods for the keypad and the display and verify that the calculator now displays its two subcomponents.*

Change the implementation of the display to show the top value of the stack. Use a table with class "keypad" containing a row with a single table data cell with *class "stackcell". Change the rendering method of the keypad to ensure that the number 0 is pushed on the stack in case it is empty. (Define and use* MyKeypad» ensureMyStackMachineNotEmpty.*) Also make it display an empty table with class "keypad". Now the calculator should display a single cell containing the value 0. If you toggle the halos, you should see something like this:*

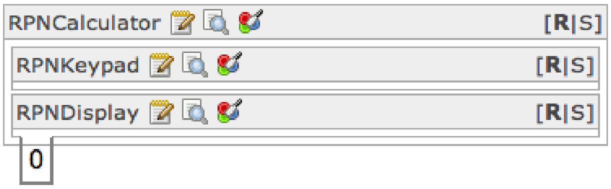


Figure 1.14: Displaying the top of the stack

Now let's implement an interface to interact with the stack. First define the following helper methods, which will make it easier to script the interface:

```
MyKeypad»renderStackButtonOn: html with: text callback: aBlock colSpan: anInteger
   html tableData
      class: 'key';
      colSpan: anInteger;
      with:
         [html anchor
            callback: aBlock;
            with: [html html: text]]
```

```
MyKeypad»renderStackButtonOn: html with: text callback: aBlock
   self
      renderStackButtonOn: html
      with: text
      callback: aBlock
      colSpan: 1
```

We will use these two methods to define the buttons on the keypad with appropriate callbacks. Certain buttons may span multiple columns, but the default is to occupy just one column.

Use the two helper methods to script the keypad as follows: (Hint: start by getting the digit and "Enter" keys working, then the arithmetic operators.)

```
MyKeypad»renderContentOn: html
  self ensureStackMachineNotEmpty.
  html table
    class: 'keypad';
    with: [
      html tableRow: [
          self renderStackButton: '+' callback: [self stackOp: #add] on: html.
          self renderStackButton: '&ndash;' callback: [self stackOp: #min] on: html.
          self renderStackButton: '&times;' callback: [self stackOp: #mul] on: html.
          self renderStackButton: '&divide;' callback: [self stackOp: #div] on: html.
          self renderStackButton: '&plusmn;' callback: [self stackOp: #neg] on: html ].
        html tableRow: [
          self renderStackButton: '1' callback: [self type: '1'] on: html.
          self renderStackButton: '2' callback: [self type: '2'] on: html.
          self renderStackButton: '3' callback: [self type: '3'] on: html.
          self renderStackButton: 'Drop' callback: [self stackOp: #pop]
            colSpan: 2 on: html ].
" and so on ... "
        html tableRow: [
          self renderStackButton: '0' callback: [self type: '0'] colSpan: 2 on: html.
          self renderStackButton: 'C' callback: [self stackClearTop] on: html.
          self renderStackButton: 'Enter'
            callback: [self stackOp: #dup. self setClearMode]
            colSpan: 2 on: html ]]
```

Check that the keypad displays properly. If you try to click on the keys, however, you will find that the calculator does not work yet ...

Implement MyKeypad»type: *to update the top of the stack by appending the typed digit. You will need to convert the top value to a string, update it, and convert it back to an integer, something like this:*

```
MyKeypad»type: aString
   stackMachine push: (stackMachine pop asString, aString) asNumber.
```

Now when you click on the digit keys the display should be updated. (Be sure that MyStackMachine»pop returns the value popped, or this will not work!)

Now we must implement MyKeypad»stackOp: *Something like this will do the trick:*

```
MyKeypad»stackOp: op
   [ stackMachine perform: op ] on: AssertionFailure do: [ ].
```

The point is that we are not sure that all operations will succeed, for example, addition will fail if we do not have two numbers on the stack. For the moment we can just ignore such errors. If we are feeling more ambitious later on, we can provide some user feedback in the error handler block.

The first version of the calculator should be working now. Try to enter some numbers by pressing the digit keys, hitting Enter *to push a copy of the current value, and entering* + *to sum the top two values.*

You will notice that typing digits does not behave the way you might expect. Actually the calculator should be aware of whether you are typing a *new* number, or appending to an existing number.

Adapt MyKeypad»type: *to behave differently depending on the current typing mode. Introduce an instance variable* mode *which takes on one of the three values* typing *(when you are typing),* push *(after you you have performed a calculator operation and typing should force the top value to be pushed), or* clear *(after you have performed* Enter *and the top value should be cleared before typing). The new* type: *method might look like this:*

```
MyKeypad»type: aString
   self inPushMode ifTrue: [
      stackMachine push: stackMachine top.
      self stackClearTop ].
   self inClearMode ifTrue: [ self stackClearTop ].
   stackMachine push: (stackMachine pop asString, aString) asNumber.
```

Typing might work better now, but it is still frustrating not to be able to see what is on the stack.

Define MyDisplayStack *as a subclass of* MyDisplay. *Add a button to the rendering method of* MyDisplay *which will call a new instance of* MyDisplayStack. *You will need an html anchor that looks something like this:*

```
html anchor
   callback: [ self call: (MyDisplayStack new setMyStackMachine: stackMachine)];
   with: 'open'
```

The callback will cause the current instance of MyDisplay to be temporarily replaced by a new instance of MyDisplayStack whose job it is to display the complete stack. When this component signals that it is done (*i.e.*, by sending self answer), then control will return to the original instance of MyDisplay.

Define the rendering method of MyDisplayStack *to display all of the values on the stack. (You will either need to define an accessor for the stack machine's* contents *or you can define* MyStackMachine»do: *to iterate over the stack values.) The stack display should also have a button labelled "close" whose callback will simply perform* self answer.

```
html anchor
  callback: [ self answer];
  with: 'close'
```

Now you should be able to *open* and *close* the stack while you are using the calculator.

There is, however, one thing we have forgotten. Try to perform some operations on the stack. Now use the "back" button of your browser and try to perform some more stack operations. (For example, open the stack, type 1, Enter twice and +. The stack should display "2" and "1". Now hit the "back" button. The stack now shows three times "1" again. Now if you type + the stack shows "3". Backtracking is not yet working.

Implement MyCalculator»states *to return the contents of the stack machine. Check that backtracking now works correctly!*

Sit back and enjoy a tall glass of something cool!

1.5 A quick look at AJAX

AJAX (Asynchronous JavaScript and XML) is a technique to make web applications more interactive by exploiting JavaScript functionality on the client side.

Two well-known JavaScript libraries are Prototype (http://www.prototypejs.org) and script.aculo.us (http://script.aculo.us). Prototype provides a framework to ease writing JavaScript. script.aculo.us provides some additional features to support animations and drag-and-drop on top of Prototype. Both frameworks Alex ▶*are?*◀ supported in Seaside through the package "Scriptaculous".

All ready-made images have the Scriptaculous package extensions already loaded. The latest version is available from http://www.squeaksource.com/Seaside. An online demo is available at http://scriptaculous.seasidehosting.st. Alternatively, if you have a enabled image running, simply go to http://localhost:8080/seaside/tests/scriptaculous.

The Scriptaculous extensions follow the same approach as Seaside itself — simply configure Smalltalk objects to model your application, and the needed Javascript code will be generated for you.

Let us look at a simple example of how client-side Javascript support can make our RPN calculator behave more naturally. Currently every keystroke to enter a digit generates a request to refresh the page. We would like instead to handle editing of the display on the client-side by updating the display in the existing page.

To address the display from JavaScript code we must first give it a unique id. Update the calculator's rendering method as follows:[12]

```
MyCalculator»renderContentOn: html
  html div id: 'keypad'; with: keypad.
  html div id: 'display'; with: display.
```

To be able to re-render the display when a keyboard button is pressed, the keyboard needs to know the display component. Add a display *instance variable to* MyKeypad*, an initializer method* MyKeypad»setDisplay:*, and call this from* MyCalculator>>initialize. *Now we are able to assign some JavaScript code to the buttons by updating* MyKeypad»renderStackButtonOn: *as follows:*

```
MyKeypad»renderStackButton: text callback: aBlock colSpan: anInteger on: html
  html tableData
    class: 'key';
    colSpan: anInteger;
    with: [
      html anchor
        callback: aBlock;
        onClick:              "handle Javascript event"
          (html updater
            id: 'display';
            callback: [ :r |
              aBlock value.
              r render: display ];
            return: false);
        with: [ html html: text ] ]
```

onClick: specifies a JavaScript event handler. html updater returns an instance of SUUpdater, a Smalltalk object representing the JavaScript Ajax.Updater object (http://www.prototypejs.org/api/ajax/updater). This object performs an AJAX request and updates a container's contents based on the response text. id: tells the updater what XHTML DOM element to update, in this case the contents of the DIV element with the id 'display'. callback: specifies a block that is triggered when the user presses the button. The

[12]If you have not implemented the tutorial example yourself, you can simply load the complete example from http://www.squeaksource.com/SqueakByExample and apply the suggested changes to the classes RPN* instead of My*.

block argument is a new renderer r, which we can use to render the display component. (Note: Even though html is still accessible, it is not valid anymore at the time this callback block is evaluated). Before rendering the display component we evaluate aBlock to perform the desired action.

return: false tells the JavaScript engine to not trigger the original link callback, which would cause a full refresh. We could instead remove the original anchor callback:, but like this the calculator will still work even if JavaScript is disabled.

Try the calculator again, and notice how a full page refresh is triggered every time you press a digit key. (The URL of the web page is updated at each keystroke.)

Although we have implemented the client-side behavior, we have not yet activated it. Now we will enable the Javascript event handling.

Click on the Configure *button in the toolbar of the calculator. Select "Add Library:"* SULibrary, *click the* Add *button and* Close.

Instead of manually adding the library, you may also do it programmatically when you register the application:

```
MyCalculator class»initialize
   (self registerAsApplication: 'rpn')
      addLibrary: SULibrary}}
```

Try the revised application. Note that the feedback is much more natural. In particular, a new URL is not generated with each keystroke.

You may well ask, *yes, but how does this work?* Figure 1.15 shows how the RPN applications would both without and with AJAX. Basically AJAX short-circuits the rendering to *only* update the display component. Javascript is responsible both for triggering the request and updating the corresponding DOM element. Have a look at the generated source-code, especially the JavaScript code:

```
new Ajax.Updater(
   'display',
   'http://localhost/seaside/RPN+Calculator',
   {'evalScripts': true,
     'parameters': ['_s=zcdqfonqwbeYzkza', '_k=jMORHtqr','9'].join('&')});
return false
```

For more advanced examples, have a further look at http://localhost:8080/seaside/tests/scriptaculous.

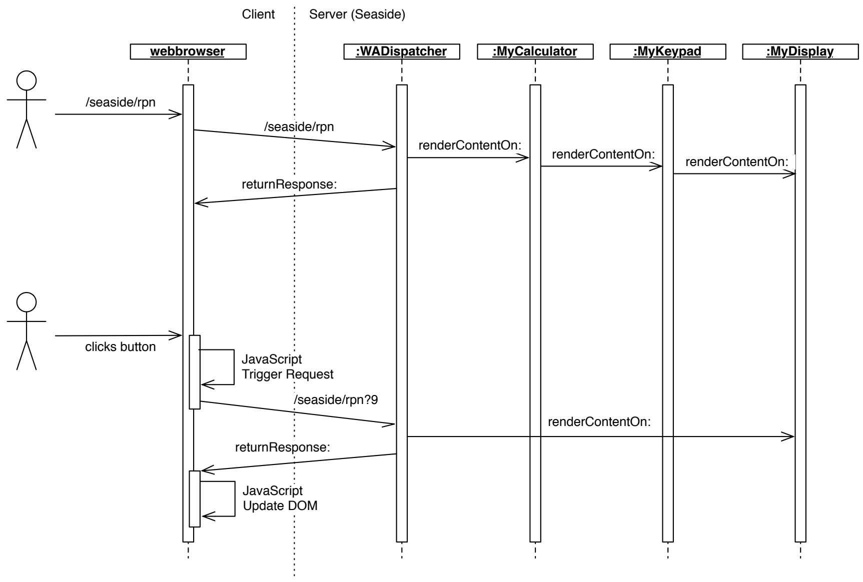


Figure 1.15: Seaside AJAX processing (simplified)

Hints. In case of server side problems use the Smalltalk debugger. In case of client side problems use FireFox (http://www.mozilla.com) with the JavaScript debugger FireBug (http://www.getfirebug.com/) plugin enabled.

1.6 Chapter summary

- The easiest way to get started is to download the "Seaside One-Click Experience" from http://seaside.st
- Turn the server on and off by evaluating WAKom startOn: 8080 and WAKom stop.
- Reset the administrator login and password by evaluating WADispatcherEditor initialize.
- Toggle Halos to directly view application source code, run-time objects, CSS and XHTML.
- Send WAGlobalConfiguration setDeploymentMode to hide the toolbar.
- Seaside web applications are composed of components, each of which is an instance of a subclass of WAComponent.

- Only a root component may be registered as a component. It should implement canBeRoot on the class side. Alternatively it may register itself as an application in its class-side initialize method by sending self registerAsApplication: *application path*. If you override description it is possible to return a descriptive application name that will be displayed in the configuration editor.

- To backtrack state, a component must implement the states method to return an array of objects whose state must be restored if the user clicks the browsers "back" button.

- A component renders itself by implementing renderContentOn:. The argument to this method is an XHTML rendering *canvas* (usually called html).

- A component can render a subcomponent by sending self render: *subcomponent*.

- XHTML is generated programmatically by sending messages to *brushes*. A brush is obtained by sending a message, such as paragraph or DIV to the html canvas.

- If you send a cascade of messages to a brush including the message with:, then this should be the last message sent. This sets the content and renders the result.

- Actions should only appear in callbacks. You should not change the state of the application while you are rendering it.

- You can bind various form widgets and anchors to instance variables with accessors by sending the message on: *instance variable* of: *object* to the brush.

- You can define the CSS for a component hierarchy by defining the method style, which should return a string containing the style sheet. (For deployed applications, it is more usual to refer to a style sheet located at a static URL.)

- Control flows can be programmed by sending x call: y, in which case component x will be replaced by y until y answers by sending answer: with a result in a callback. The receiver of call: is usually self, but may in general be any visible component.

- A control flow can also be specified as a *task* — a instance of a subclass of WATask. It should implement the method go, which should call a series of components in a workflow.

- Use WAComponents's convenience methods request:, inform:, confirm: and chooseFrom:caption: for basic interactions.
- To prevent the user from using the browser's "back" button to access a previous execution state of the web application, you can declare portions of the workflow to be a *transaction* by enclosing them in an isolate: block.