

Chapter 1

Seaside by Example

Seaside is a framework for building web applications in Smalltalk. It was originally developed by Avi Bryant in 2002; once mastered, Seaside makes web applications almost as easy to write as desktop application.

Two of the better known applications built with Seaside are Squeak-Source¹ and Dabble DB². 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. Instead, you just send messages to objects. What a nice idea!

1.1 Why do we need Seaside?

Modern web applications try to interact with the user in the same way as desktop applications: they ask the user questions, and the user responds, usually by filling in a form or clicking a button. But the web works the other way around: the user's browser makes a request of the server, and the server responds with a new web page. So web application development frameworks have to cope with a host of problems, chief among them being the management of this "inverted" control flow. Because of this, many web applications try to forbid the use of the browser's "back" button due to the difficulty of keeping track of the state of a session. Expressing non-trivial control flows across multiple web pages is often cumbersome, and multiple control flows can be difficult or impossible to express.

Seaside is a component-based framework that makes web development easier in several ways. First, control flow can be expressed naturally using

¹<http://SqueakSource.com>

²<http://DabbleDB.com>

message sends. Seaside keeps track of which web page corresponds to which point in the execution of the web application. This means that the browser's "back" button works correctly.

Second, state is managed for you. As the developer, you have the choice of enabling backtracking of state, so that navigation "back" in time will undo side-effects. Alternatively, you can use the transaction support built into Seaside to prevent users from undoing permanent side-effects when they use the back button. You do not have to encode state information in the URL — this too is managed automatically for you.

Third, web pages are built up from nested components, each of which can support its own, independent control flow. There are no XHTML templates — instead valid XHTML is generated programmatically using a simple Smalltalk protocol. 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 site³. This is a prepackaged version of Seaside 2.8 for Mac OS X, 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.

If you are feeling more adventurous, an alternative to the "one-click" image is to start with the latest Squeak Developers' Web image⁴, and install Seaside yourself by following the manual installation instructions on the Seaside web site.

Seaside includes a web server; you can turn the server on, telling it to listen on port 8080, by evaluating `WAKom startOn: 8080`, and you can turn it off again by evaluating `WAKom stop`. In the default installation, the default administrator login is `admin` and the default password is `seaside`. To change them, evaluate: `WADispatcherEditor initialize`. This will prompt you for a new name and password.

³<http://seaside.st>

⁴<http://damien.cassou.free.fr/squeak-dev.html>

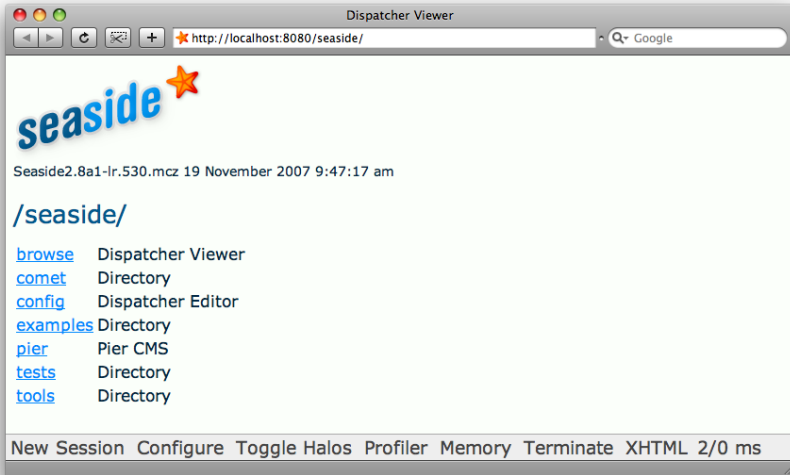



Figure 1.1: Starting up Seaside

 Start the Seaside server and direct a web browser to `http://localhost:8080/seaside/`.

You should see a web page that looks like Figure 1.1.

 Navigate to the `examples` ▶ `counter` page. (Figure 1.2)

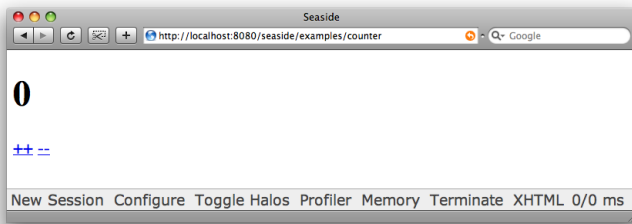



Figure 1.2: The counter.

This page is a small Seaside application: it displays 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 currently displayed state, rather than the state that the counter was in when you started using the “back” button.

Notice the toolbar at the bottom of the web page in Figure 1.1. Seaside supports a notion of “sessions” to keep track of the state of the application for different users. 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 a way to explore the state of the application running on the Seaside server. Profiler and Memory provide detailed information about the run-time performance of the application. XHTML can be used to validate the generated web page, but works only when the web page is publicly accessible from the Internet, because it uses the W3C validation service.

Seaside applications are built up from pluggable “components”. In fact, components are ordinary Smalltalk objects. The only thing that is special about them is that they should be instances of classes that inherit from the Seaside framework class `WACComponent`. We can explore components and their classes from the Squeak image, or directly from the web interface using halos.

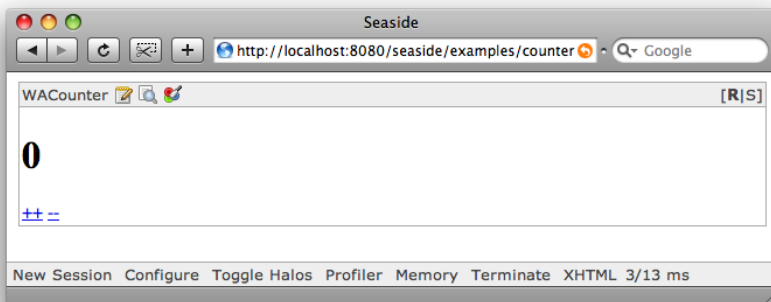



Figure 1.3: Halos

 Select Toggle Halos. You should see a web page like Figure 1.3. At the top left the text `WACounter` tells us the class of the Seaside component that implements the behavior of this web page. Next to this are three clickable icons. The first, with the pencil, activates a Seaside class browser on this class. The second, with the

magnifying glass, opens an object inspector on the currently active `WACounter` instance. The third, with the coloured circles, displays the CSS style sheet for this component. At the top right, the **R** and **S** let you toggle between the rendered and source views of the web page. Experiment with all of these links. Note that the `++` and `-` links are also active in the source view. Contrast the nicely-formatted source view provided by the Halos with the unformatted source view offered by your browser.

The Seaside class browser and object inspector can be very convenient when the server is running on another computer, especially when the server does not have a display, or if it is in remote place. However, when you are first developing a Seaside application, the server will be running locally, and it is easy to use the ordinary Squeak development tools in the server image.

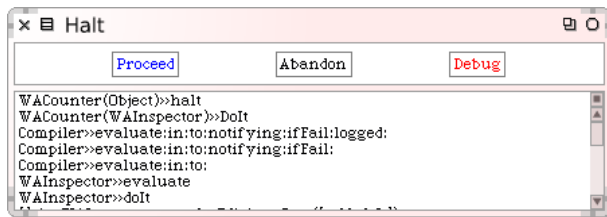


Figure 1.4: Halting the counter

*Using the object inspector link in the web browser, open an inspector on the underlying Smalltalk counter object and evaluate `self halt`. The web page will stop loading. Now switch to the Seaside image. You should see a pre-debugger window (Figure 1.4) showing a `WACounter` object executing a `halt`. Examine this execution in the debugger, and then **Proceed**. Go back to the web browser and notice that the counter application is running again.*

Seaside components can be instantiated multiple times, and in different contexts.

Point your web browser to <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 of `WAMultiCounter`. You should see three methods on the class side (`canBeRoot`, `description`, and `initialize`) and three on the instance side (`children`, `initialize`, and `renderContentOn:`). Note that an application is simply a component that is willing to be at the root of the

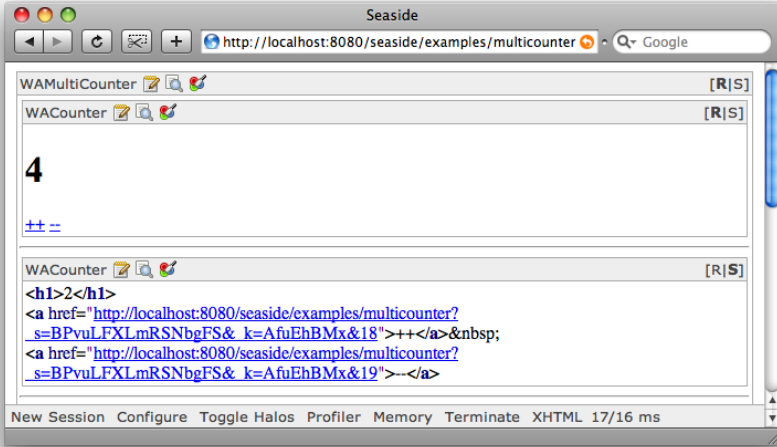



Figure 1.5: Independent subcomponents

component containment hierarchy; this willingness is indicated by defining a class-side method `canBeRoot` to answer `true`.

You can use the Seaside web interface to configure, copy or remove individual applications (i.e., root-level components). Try making the following configuration change.

 Point your web browser to `http://localhost:8080/seaside/config`. Supply the login and password (admin and seaside by default). Select `Configure` next to "examples." Under the heading "Add entry point", enter the new name "counter2" for the type `Application` and click on `Add` (see Figure 1.6). On the next screen, set the Root Component to `WACounter`, then click `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.

Seaside operates in two modes: *development* mode, which is what we have seen so far, and *deployment* mode, in which the toolbar is not available. You can put Seaside into deployment mode using either the configuration page `Andrew ▶ How? I couldn't find this ◀` or the `Configure` button in the toolbar. In either case, set the deployment mode to `true`. Note that this affects new sessions only. You can also set the mode globally by evaluating `WAGlobalConfiguration setDeploymentMode` or `WAGlobalConfiguration setDevelopmentMode`.

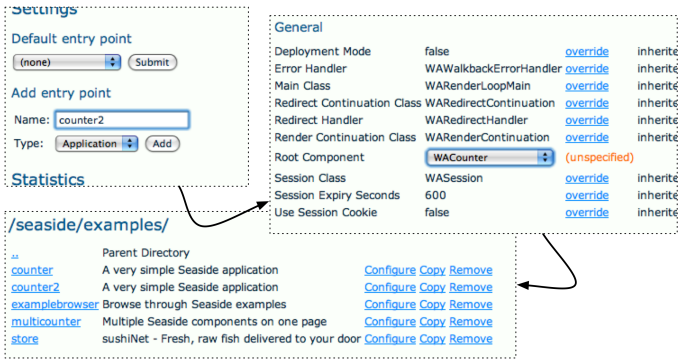


Figure 1.6: Configuring a new application

The configuration web page is just another Seaside application, so it too can be controlled from the configuration page. If you remove the “config” application, you can get it back by evaluating WADispatcherEditor initialize.

1.3 Seaside components


Andrew ▶ This section was too long — 18 pages. It also contained several self-references (“see section 1.3”). So I broke into smaller sections, by promoting some of the subsections and subsubsections. ◀

As we mentioned in the previous section, Seaside applications are built out of *components*. Let’s take a closer look at how Seaside works by implementing the *Hello World* component.

Every Seaside component should inherit directly or indirectly from WAComponent, as shown in Figure 1.8.

 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, which knows how to render XHTML.

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


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

Now we must inform Seaside that this component is willing to be a standalone application.

 Implement the following method on the class side of WAHelloWorld.

```
WAHelloWorld class>canBeRoot
  ↑ true
```

We are almost done!

 Point your web browser at <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! You should see a web page like Figure 1.7.

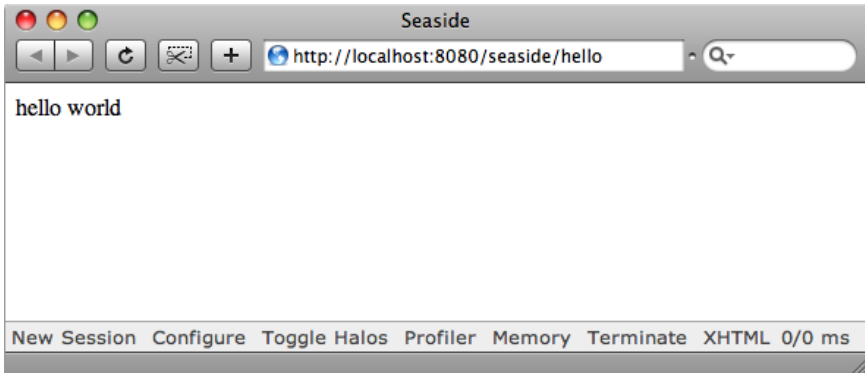


Figure 1.7: “Hello World” in Seaside

State backtracking and the “Counter” Application

The “counter” application is only slightly more complex than the “hello world” application.

The class WACounter is a standalone application, so WACounter class must answer true to the canBeRoot message. It must also register itself as an application; this is done in its class-side initialize method, as shown in Figure 1.8.

WACounter defines two methods, increase and decrease, which will be triggered from the ++ and -- links on the web page. It also defines an instance variable count to record the state of the counter. However, we also want Seaside to synchronize the counter with the browser page: when the user clicks on the browser’s “back” button, we want seaside to “backtrack”

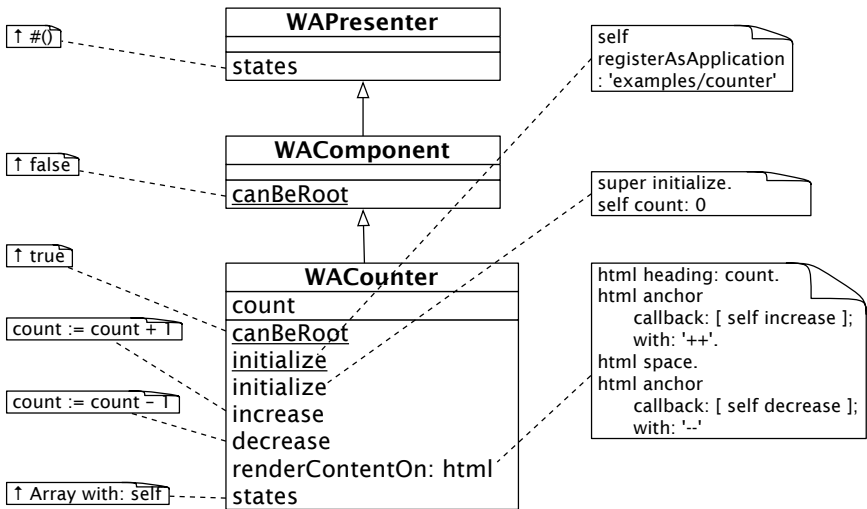


Figure 1.8: The WACounter class, which implements the *counter* application. Methods with underlined names are on the class-side; those with plain-text names are on the instance side.

the state of the WACounter object. Seaside includes a general mechanism for backtracking, but each application has to tell Seaside which parts of its state to track.

A component enables backtracking by implementing the `states` method on the instance side: `states` should answer an array containing all the objects to be tracked. In this case, the WACounter object adds itself to Seaside's table of backtrackable objects by returning `Array with: self`.

Caveat. There is a subtle but important point to watch for when declaring objects for backtracking. Seaside tracks state by making a *copy* of all the objects declared in the `states` array. It does this using a `WASnapshot` object; `WASnapshot` is a subclass of `IdentityDictionary` that records the objects to be tracked as keys and shallow copies of their state as values. If the state of an application is backtracked to a particular snapshot, the state of each object entered into the snapshot dictionary is overwritten by the copy saved in the snapshot.

Here is the point to watch out for: In the case of WACounter, you might think that the state to be tracked is a number — the value of the `count` instance variable. However, having the `states` method answer `Array with: count`

won't work. This is because the object named by `count` is an integer, and integers are immutable. The `increase` and `decrease` methods don't change the state of the object 0 into 1 or the object 3 into 2. Instead, they make `count` name a different integer: every time the count is incremented or decremented, the object named by `count` is *replaced* by another. This is why `WACounter»states` must return `Array with: self`. When the state of a `WACounter` object is replaced by a previous state, the *value* of each of the instance variable in the object is replaced by a previous value; this correctly replaces the current value of `count` by a prior value.

1.4 Rendering XHTML

The purpose of a web application is to create, or “render”, web pages. As we mentioned in Section 1.3, each Seaside component is responsible for rendering itself. So, let's start our exploration of rendering by seeing how the counter component renders itself.

Rendering the Counter

The rendering of the counter is relatively straightforward; the code is shown in Figure 1.8. The current value of the counter is displayed as an XHTML heading, and the increment and decrement operations are implemented as html anchors (that is, links) with callbacks to blocks that will send `increase` and `decrease` to the counter object.

We will have a closer look at the rendering protocol in a moment. But before we do, let's have a quick look at the multi-counter.

From Counter to MultiCounter

`WAMultiCounter`, shown in Figure 1.9 is also a standalone application, so it overrides `canBeRoot` to answer `true`. In addition, it is a *composite* component, so Seaside requires it to declare its children by implementing a method `children` that answers an array of all the components it contains. It 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!

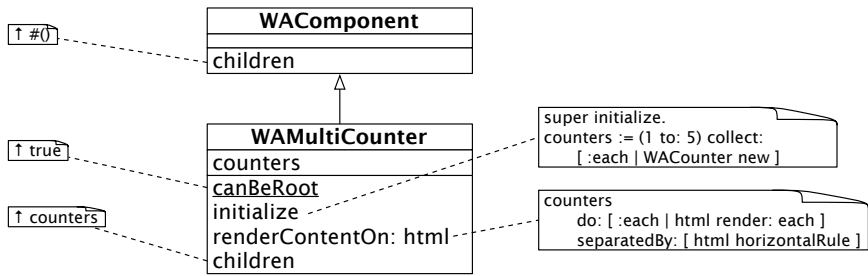


Figure 1.9: WAMultiCounter

More about Rendering XHTML

As you can see from these examples, Seaside does not use templates to generate web pages. Instead it generates XHTML programmatically. The basic idea is that every Seaside component should override the method `renderContentOn:`; this message will be sent by the framework to each component that needs to be rendered. This `renderContentOn:` message will have argument that is an *html canvas* onto which the component should render itself. By convention, the *html canvas* parameter is called `html`. An *html canvas* is analogous to the *graphics canvas* used by Morphic (and most other drawing frameworks) to abstract away from the device-dependent details of drawing.

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: anyObject` can be sent to an *html canvas* to render `anyObject`; it is normally used to render subcomponents. `anyObject` will itself be sent the message `renderContentOn:` this is what happens in `multi-counter` (see Figure 1.9).

Using Brushes

A *canvas* provides a number of *brushes* that can be used to render (*i.e.*, “paint”) content on the canvas. There are brushes for every kind of XHTML element — paragraphs, tables, lists, and so on. To see the full protocol of brushes and convenience methods, you should browse the class `WACanvas` and its subclasses. The argument to `renderContentOn:` 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.

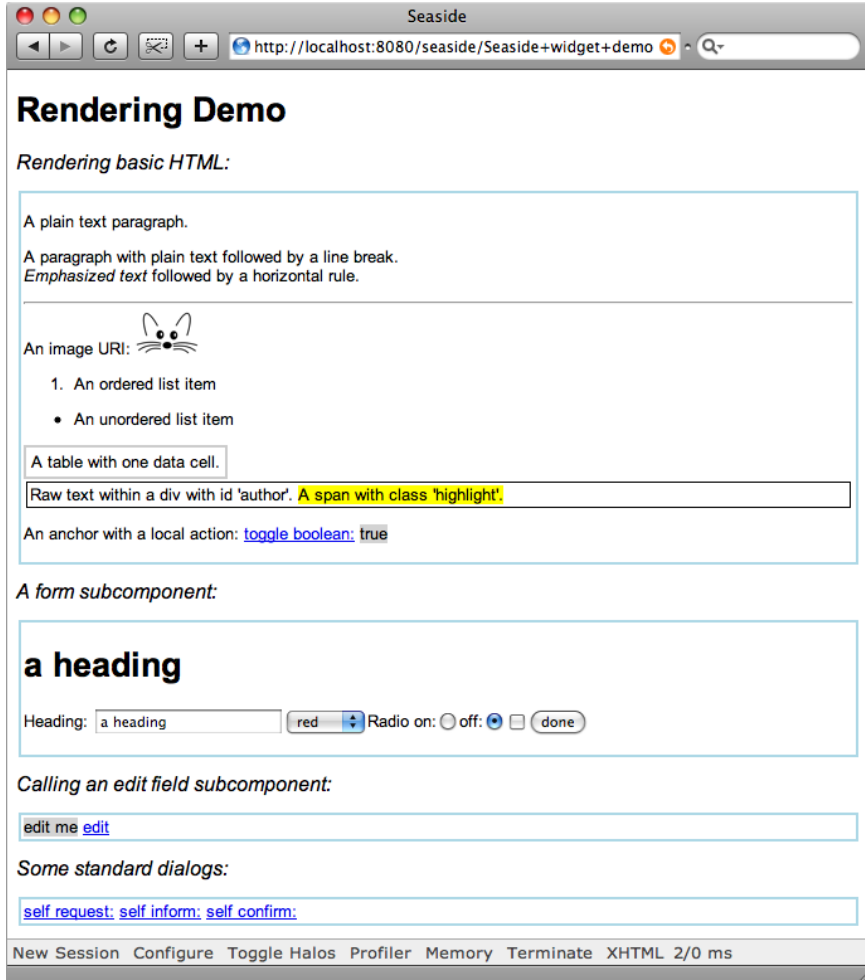


Figure 1.10: RenderingDemo

In Figure 1.10 we can see the output of many of the basic brushes

offered by Seaside.⁵ The root component SeasideDemo simply renders its subcomponents, which are instances of SeasideHtmlDemo, SeasideFormDemo, SeasideEditCallDemo and SeasideDialogDemo, as shown in method 1.1.

Method 1.1: SeasideDemo»renderContentOn:

```
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 ..."
```

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

```
SeasideDemo»children
  ↑ { htmlDemo . formDemo . editDemo . dialogDemo }
```

Notice that there are two different ways of instantiating the heading brush. The first way is to set the text directly by sending the message heading:. The second way is instantiate the brush by sending heading, and then to send a cascade of messages to the brush to set its properties and render it. Many of the available brushes can be used in these two ways.

If you send a cascade of messages to a brush including the message with:, then with: should be the *final* message. with: both sets the content and renders the result.

In method 1.1, the first heading is at level 1, since this is the default. 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 1.5.) Also note that the argument to the with: keyword message need not be a literal string: it can 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
```

⁵The source code for method 1.1 is in the package SBE-SeasideDemo in the project <http://www.squeaksource.com/SqueakByExample>.

```
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 using the pattern `render-On:`. All rendering methods go in the *rendering* protocol. Don't send `renderContentOn:` from your own code, use `render:` instead.

Look at the following code. The first helper method, `SeasideHtmlDemo»renderParagraphsOn:`, shows you how to generate XHTML paragraphs, plain and emphasized text, and images. Note that in Seaside simple elements are rendered by specifying the text they contain directly, whereas complex elements are specified using blocks. This is a simple convention to help you structure your rendering code.

```
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, `SeasideHtmlDemo»renderListsAndTablesOn:`, shows you how to generate lists and tables. A table 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 how these XHTML elements should be displayed (see Section 1.5).

```
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” (*i.e.*, a link). Clicking on the link will cause the subsequent text to toggle between “true” and “false” by toggling the instance variable `toggleValue`.

```
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 ]
```

Note that actions should appear only in callbacks. The code executed while rendering should not change the state of the application!

Forms

Forms are rendered just like the other examples that we have already seen. Here is the code for the `SeasideFormDemo` component in Figure 1.10.

```

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. Note that all the state changes happen in the callbacks, not as part of the rendering.

There is one Seaside feature used here that is worth special mention, namely the message `on:of:`. In the example, this message is used to bind a text input field to the variable `heading`. Anchors and buttons also support this message. The first argument is the name of an instance variable for which accessors have been defined; the second argument is the object to which this instance variable belongs. Both observer (`heading`) and mutator (`heading:`) accessor messages must be understood by the object, with the usual naming convention. In the case here of a text input field, this saves us the trouble of having to define a callback that updates the field as well as having to bind the default contents of the html input field to the current value of the instance variable. Using `on: #heading of: self`, the `heading` variable is updated automatically whenever the user updates the text input field.

The same message is used twice more in this example, to cause the selection of a colour on the html form to update the color variable, and to bind the result of the checkbox to the checked variable. Many other examples can be found 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 browse the source code of the examples directly using the Seaside class browser.

1.5 CSS: Cascading style sheets

Cascading Style Sheets⁶, 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 `WAFFileLibrary`, 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.

Instead of directly encoding display attributes in the paragraph and text elements of your web pages, with CSS you will define different classes of elements and place all display considerations in a separate style sheet. Paragraph-like entities are called *divs* and text-like entities are *spans*. You would then define symbolic names, like "highlight" (see example below) for text to be highlighted, and specify how highlighted text is to be displayed in your style sheet.

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 *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 1.11 illustrates a simple style sheet for the rendering demo shown earlier in Figure 1.10. 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 (h2), tables (table), and table data (td).

The remaining rules have selectors that will match XHTML elements that have the given "class" or "id" attributes. CSS selectors for class at-

⁶<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 1.11: SeasideDemoWidget common style sheet.

tributes start with a “.” and those for id attributes with “#”. The main difference between class and id attributes is that many elements may have the same class, but only one element may have a given id (*i.e.*, an *identifier*). So, whereas a class attribute, such as `highlight`, may occur multiple times on any page, an id must identify a *unique* element on the page, such as a particular menu, the modified date, or author. Note that a particular XHTML element may have multiple classes, in which case all the applicable display attributes will be applied in sequence.

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 match a span within a paragraph but not within a div.

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 Garden⁷, which shows how the same content can be rendered in radically different ways simply by changing the CSS style sheet.

1.6 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 can *call* another component by sending `call: callee`. The caller is temporarily replaced by the callee, until the callee returns control by sending `answer:`. The caller is usually *self*, but could also be any other currently visible component.
2. A workflow can be defined as a *task*. This is a special kind of component that subclasses `WATask` (instead of `WACComponent`). Instead of defining `renderContentOn:`, it defines no content of its own, but rather defines a `go` method that sends a series of `call: message`s to activate various subcomponents in turn.

Call and answer

Call and answer are used to realize simple dialogues.

There is a trivial example of `call:` and `answer:` in the rendering demo of Figure 1.10. The component `SeasideEditCallDemo` displays a text field and an *edit* link. The callback for the edit link calls a new instance of `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.

(We underline the `call:` and `answer:` sends to draw attention to them.)

```
SeasideEditCallDemo>renderContentOn: html
html span
  class: 'field';
  with: self text.
html space.
html anchor
```

⁷<http://www.csszengarden.com/>

```
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 the new web page that must be created. At run-time, a new page is created in which the `SeasideEditCallDemo` component is replaced by a `SeasideEditAnswerDemo` component; the parent component and the other peer components are untouched.

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

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`. The generated dialogues are shown in Figure 1.12. We can see these 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. An optional label and default value may also be specified.`

```
SeasideDialogDemo>renderContentOn: html
  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: ':-(' ]
```

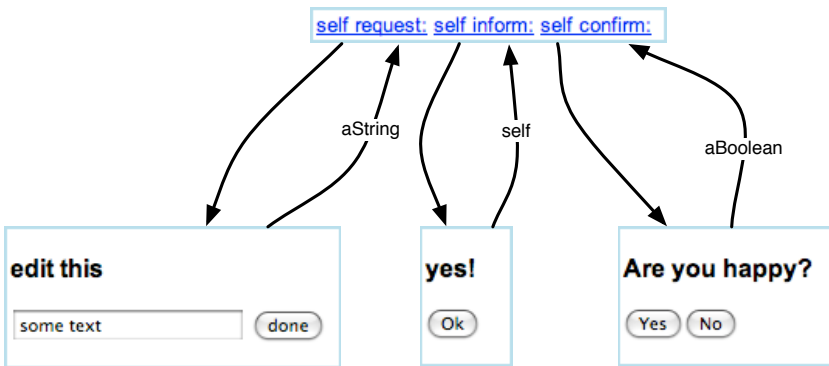


Figure 1.12: Some standard dialogs

```
];
with: 'self confirm:'.
```

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

Tasks

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, generated by the convenience method `chooseFrom:caption:`, is a `WAChoiceDialog` that asks the user to choose a cheese.

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

The second is a `WAYesOrNoDialog` to confirm the choice (generated by the convenience method `confirm:`).

```
WAConvenienceTest>confirmCheese
↑self confirm: 'Is ', cheese, ' your favorite cheese?'
```

Finally a `WAFORMDialog` is called (via the convenience method `inform:`).

```
WAConvenienceTest>informCheese
self inform: 'Your favorite cheese is ', cheese, '!
```

The generated dialogues are shown in Figure 1.13.

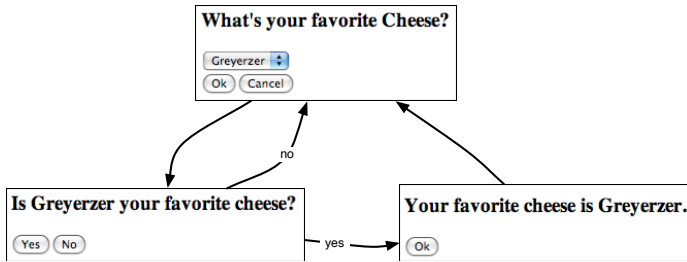


Figure 1.13: A simple task

Transactions

We saw in Section 1.3 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: all that a component need do is implement the method `states` to answer an array of all the objects whose state must be tracked.

However, sometimes, we do not want to backtrack state: instead we want to *prevent* the user from accidentally undoing effects that should be permanent. This is often referred to as “the shopping cart problem”. Once you have checked-out your shopping cart 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 most recent valid page.

The Seaside *Sushi Store* is sample application that illustrates many of the features of Seaside, including transactions. This application is bundled with your installation of Seaside, so you can try it out by pointing your browser at <http://localhost:8080/seaside/examples/store>.⁸

The sushi store supports the following workflow:

1. Visit the store.
2. Browse or search for sushi.

⁸If you cannot find it in your image, there is a version of the sushi store available on SqueakSource from <http://www.squeaksource.com/SeasideExamples/>.

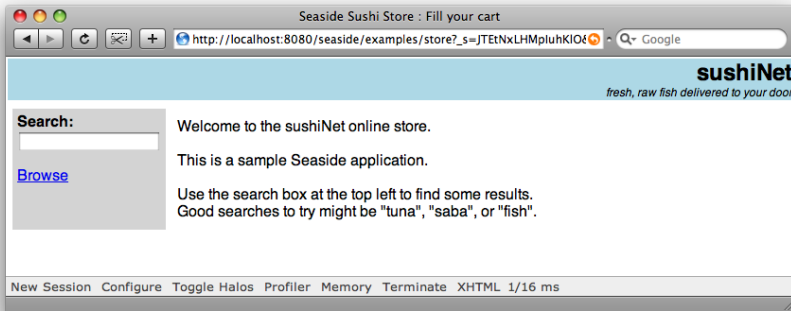



Figure 1.14: The Sushi Store

3. Add sushi to your shopping cart.
4. Checkout.
5. Verify your order.
6. Enter shipping address.
7. Verify shipping address.
8. Enter payment information.
9. Your fish is on its way!

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

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

`WASStoreTask` captures this workflow sequence. At a couple of points it is critical that the user not be able to go back and change the submitted information.

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

Seaside lets the programmer say that a certain part of a workflow act like a transaction: once the transaction is complete, the user cannot go back and undo it. You say this by sending `isolate:` to a task with the transactional

block as its argument. We can see this in the sushi store workflow as follows:

```

WASoreTask»go
| shipping billing creditCard |
cart := WASoreCart 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. However, once both transactions are complete, any attempt to navigate back will fail.

Transactions may also be nested. A simple demonstration of this is found in the class `WANestedTransaction`. The first `isolate:` takes as argument a block that contains another, nested `isolate:`

```

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.

1.7 A complete tutorial example

Let's see how we can build a complete Seaside application from scratch.⁹ 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. The calculator with the two display options is shown in Figure 1.15.

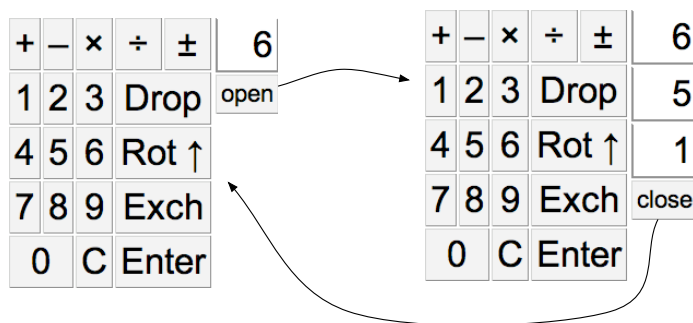




Figure 1.15: RPN calculator and its stack machine

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:

⁹The 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.¹⁰ 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:


- `MyRPNWidget` — 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 `WACComponent` and the direct superclass of the following 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`.

¹⁰It's a good idea to use `Object»assert:` to specify the preconditions for an operation. This method will raise an `AssertionFailure` if the user tries to use the stack machine in an invalid state.


 Define `MyRPNWidget` 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.

```
MyRPNWidget»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 `MyRPNWidget`. 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, `MyRPNWidget`. 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.

Now we will 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:

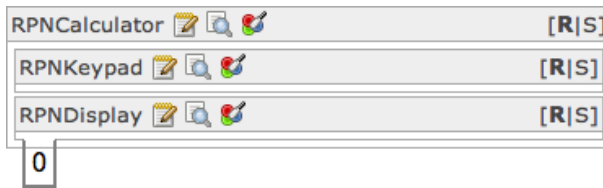



Figure 1.16: Displaying the top of the stack


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!

1.8 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 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:¹¹

```
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


¹¹If 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 link 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 1.17 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>.

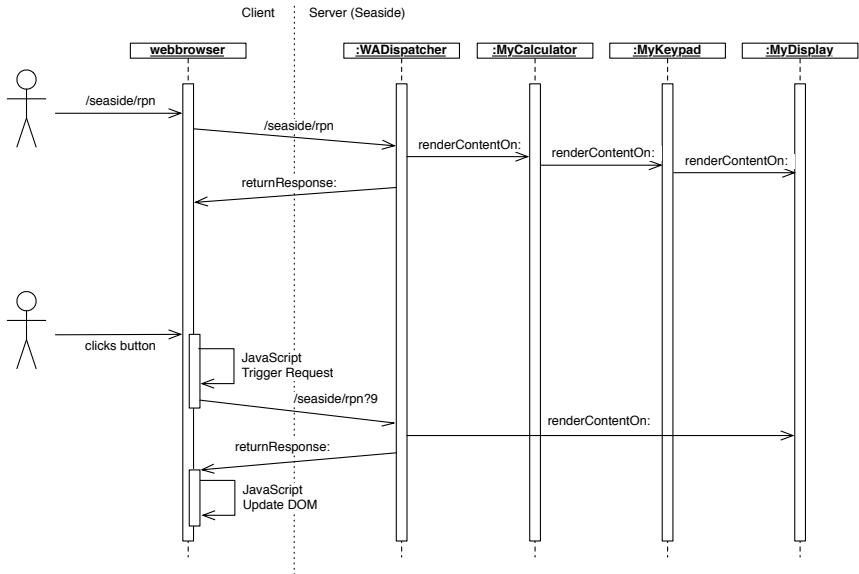


Figure 1.17: Seaside AJAX processing (simplified)

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.

1.9 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 answer an array of objects whose state will be restored if the user clicks the browser's "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: sub-component`.
- 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 that includes the message `with:`, then `with:` should be the last message sent. The `with:` message sets the contents *and* renders the result.
- Actions should appear only 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*—an 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`.