Some of the key tools of the Pharo environment

The goal of this chapter is to present the main tools of the Pharo programming environment. You have already seen how to define methods and classes using the browser; this chapter will show you more of its features, and introduce you to some of the other browsers.

Of course, very occasionally you may find that your program does not work as you expect. Pharo has an excellent debugger, but like most powerful tools, it can be confusing on first use. We will walk you through a debugging session and demonstrate some of the features of the debugger.

One of the unique features of Pharo (and its ancestors) is that while you are programming, you are living in a world of live objects, not in a world of static program text. This makes it possible to get very rapid feedback while programming, which makes you more productive. There is a tool that let you look at, and indeed change, live objects: the inspector.

1.1 Pharo environment overview

The System Browser is the central development tool. You will use it to create, define, and organize your classes and methods. Using it you can also navigate through all the library classes. Unlike other environments where the source code is stored in separate files, in Pharo all classes and methods are contained in the image.

The Finder tool will let you find methods, classes, pragmas, and more. You can look for a method’s name, a class name, some source code, a pragma’s
name or even look for methods by providing an example!

The Monticello Browser is the starting point for loading code from, and saving code in, Monticello packages. It is discussed in more detail in Chapter: Sharing Code and Source Control.

The Process Browser provides a view of all of the processes (threads) executing in Smalltalk.

The Test Runner lets you run and debug SUnit tests, and is described in more detail in Chapter: SUnit.

The Transcript is a window on the Transcript output stream, which is useful for writing log messages.

The Playground is a window into which you can type input. It can be used for any purpose, but is most often used for typing Pharo expressions and executing them via Do it. We have already briefly encountered the Playground (and the Transcript) in Chapter: A Quick Tour of Pharo.

The Debugger has an obvious role, but you will discover that it has a more central place compared to debuggers for other programming languages, because in Pharo you can program in the debugger. The debugger is not launched from a menu; it is normally entered by running a failing test, by typing CMD-. to interrupt a running process, or by inserting a Halt now expression in code.

Window groups

Managing multiple windows within a Pharo image can become a tedious process. Window Groups are windows that offer tab support similar to the one you are used to on your web browser. To create a window group, click on the down arrow which appears on the top right corner of every window within Pharo, and select Create window group. This will turn that window into a window group with a tab bar, containing as its first tab the original contents of the window. You can add other windows to the group (within the Pharo image only, of course), by dragging and dropping their title bars onto the tab bar, next to existing tabs. Each new window that you drag onto the tab bar will be added as a new tab.

Themes and icon sets

If, like some of us, you are not a fan of the default bright theme, there is also a dark theme that can be used with Pharo. It can be found in the Settings browser in the World Menu (World > System > Settings), in the Appearance section. The User interface theme pulldown allows you to switch between the default Pharo3 theme and the Pharo3 Dark theme. You can also experiment with available icon sets in the next pulldown in that section.

Please note that you may have to close and reopen existing windows to redraw them correctly with the new dark theme.
1.2 The main code browser

You can use the Catalog Browser to install additional themes. Open World > Tools > Catalog Browser and search for theme in the search box. For example, if you want something more colorful, you can download the Nireas theme, which is a blue theme inspired by classic home computers like Amiga 500 and Amstrad CPC 6128. Nireas comes with a GUI tool that allows you to customize the theme’s colors to your liking, if blue is not your thing.

Of course, if you are feeling especially adventurous, you can even make your own themes using existing ones as templates.

1.2 The main code browser

Many different class browsers have been developed over the years for Pharo. Pharo simplifies this story by offering a single browser that integrates various views. Figure 1.2 shows the browser as it appears when you first open it.

The four small panes at the top of the browser represent a hierarchic view of the methods in the system, much in the same way as the Mac OS X Finder in column mode provide a view of the files on the disk. The leftmost pane lists packages of classes; select one (say Kernel) and the pane immediately to the right will then show all of the classes in that package.

Similarly, if you select one of the classes in the second pane, say, Boolean (see Figure 1.3), the third pane will show all of the protocols defined for that class, as well as a virtual protocol --all--. Protocols are a way of categorizing methods; they make it easier to find and think about the behaviour of a class by breaking it up into smaller, conceptually coherent pieces. The fourth pane shows the names of all of the methods defined in the selected protocol. If you then select a method name, the source code of the corresponding method
Figure 1.2: The main code browser.

Figure 1.3: The browser with the class Boolean selected.
1.2 The main code browser

Figure 1.4: Browsing the `or:` method in class `Boolean`.

appears in the large pane at the bottom of the browser, where you can view it, edit it, and save the edited version. If you select class `Boolean`, protocol `controlling` and the method `or:`, the browser should look like Figure 1.4.

Unlike directories in a file browser, the four top panes of the browser are not quite equal. Whereas classes and methods are part of the Smalltalk language, packages and protocols are not: they are a convenience introduced by the browser to limit the amount of information that needs to be shown in each pane. For example, if there were no protocols, the browser would have to show a list of all of the methods in the selected class; for many classes this list would be too large to navigate conveniently.

Because of this, the way that you create a new package or a new protocol is different from the way that you create a new class or a new method. To create a new package, right-click in the package pane and select `Add package`.

To create a new protocol, right-click in the protocol pane and select `Add protocol`.

Enter the name of the new thing in the dialog, and you are done: there is nothing more to a package or a protocol than its name and its contents.

In contrast, to create a new class or a new method, you will actually have to write some Smalltalk code. If you click the currently selected package (in the left-most pane), the bottom browser pane will display a class creation
Some of the key tools of the Pharo environment

Figure 1.5: Browser showing the class-creation template

template (Figure 1.5). You create a new class by editing this template: replace *Object* by the name of the existing class of which you wish to create a subclass, replace *NameOfSubclass* by the name that you would like to give to your new subclass, and fill in the instance variable names if you know them. The package for the new class is by default the currently selected package, but you can change this too if you like. If you already have the browser focused on the class that you wish to subclass, you can get the same template with slightly different initialization by right-clicking in the class pane, and selecting *Add Class*. You can also just edit the definition of an existing class, changing the class name to something new. In all cases, when you accept the new definition, the new class (the one whose name follows the #) is created (as is the corresponding metaclass). Creating a class also creates a global variable that references the class, which is why you can refer to all of the existing classes by using their names.

Can you see why the name of the new class has to appear as a *Symbol* (i.e., prefixed with #) in the class creation template, but after the class is created, code can refer to the class by using the name as an identifier (without the #)?

The process of creating a new method is similar. First select the class in which you want the method to live, and then select a protocol. The browser will display a method-creation template, as shown in Figure 1.6, which you can fill-in or edit.
1.2 The main code browser

Navigating the code space

The browser provides several tools for exploring and analyzing code. These tools can be accessed by right-clicking in the various contextual menus, or, in the case of the most frequently used tools, by means of keyboard shortcuts.

Opening a new browser window

Sometimes you want to open multiple browser windows. When you are writing code you will almost certainly need at least two: one for the method that you are typing, and another to browse around the system to see how things work. You can open a browser on a class named by any selected text using the CMD-b keyboard shortcut.

To do Try this: In a playground window, type the name of a class (for instance Morph), select it, and then press CMD-b. This trick is often useful; it works in any text window.

Message senders

While writing new code, refactoring existing code, or while trying to learn how to use unfamiliar libraries, you will frequently want to know the senders
Some of the key tools of the Pharo environment

and implementors of various messages.

There are several ways of discovering where in the codebase a message is used, by listing its senders:

1. *From the method pane.* Select a method in the method pane of the browser. You can then right-click on it and select Senders of (b,n) in the context menu. Alternatively, you can also use the shortcut CMD-b CMD-n to do the same thing (that’s what the b,n in the menu item stands for). To help remember this shortcut, think: browse senders.

2. *From the code pane.* Highlight a particular message in the source code. This can be done in a code pane of a browser, in a Playground window, or in any text window. If you want to find the senders of a keyword message, you highlight all of the keywords in the message, including arguments. Then, you can right-click on the highlighted selector and choose Code search > senders of it (n). Alternatively, you can use the shortcut CMD-n instead of right-clicking.

3. *Using Spotter.* Bring up a particular method in Spotter (press SHIFT-Enter to bring up the Spotter search box, type in the message selector, arrow down to a particular Implementor of that message, and press CMD-right arrow to focus the search on it.) A list of Senders now appears in the search results.
1.2 The main code browser

Figure 1.8: The Senders Browser showing that the Canvas>>draw method sends the drawOn: message to its argument.

Only a handful of senders are shown by default, but you can view the full list by clicking on the arrow next to the Senders category (or arrow down to the Senders list and expand it by pressing CMD-SHIFT-right arrow).

Let’s try some of these in action.

Open a browser on the Morph class, select the Morph>>drawOn: method in the method pane. If you now press CMD-b CMD-n (or right-click in the method pane and select Senders of... (Figure 1.7)), a browser will open with the list of all methods in the image that send the selected message (Figure 1.8).

Now look at the third sender on the list, Canvas>>draw:. You can see that this method sends drawOn: to whatever object is passed to it as an argument, which could potentially be an instance of any class at all. Dataflow analysis can help figure out the class of the receiver of some messages, but in general, there is no simple way for the browser to know which message-sends might cause which methods to be executed. For this reason, the Senders browser shows exactly what its name suggests: all of the senders of the message with the chosen selector. The senders browser is nevertheless extremely useful when you need to understand how you can use a method: it lets you navigate quickly through example uses. Since all of the methods with the same selector
Some of the key tools of the Pharo environment

Some of the key tools of the Pharo environment

should be used in the same way, all of the uses of a given message ought to be similar.

To do Switch to the Dictionary class in the browser (remember, you can right-click in the package or class pane and select Find class..., or just use the CMD-f CMD-c shortcut), and select the addAll: method in the method pane.

Looking at the source code, suppose you wanted to see all of the senders of the at: key put: value message. You can simply highlight the whole message send, and press CMD-n (or right-click and select Code Search > senders of it (n)), to bring up the list of senders (see Figure 1.9).

Message implementors

Similarly, you may come across a message, and want to see how it’s implemented. This is what the Implementors browser is for. It works in the same way as the Senders browser, but instead lists all of the classes that implement a method with the same selector.

1. From the method pane. Select a method in the method pane. You can then bring up the Implementors browser by right-clicking on the method and selecting Implementors of...(b,m) in the context menu (or use the shortcut CMD-b CMD-m). To help remember this shortcut, think: browse implementors.

2. From the code pane. Highlight a particular message in the source code (or any text window). If you want to find the implementors of a keyword message, you highlight all of the keywords in the message. Then, you can right-click on the highlighted selector and choose Code search... > implementors of it (m) from the menu (or just use the shortcut CMD-n).

3. Using Spotter. Bring up a method in Spotter (press SHIFT-Enter to bring up the Spotter search box, and start typing the message selector). The Implement-
tors category will show up in the search results, showing the first handful of implementors. To see the full list, click on the arrow to the right of Implementors category (or arrow down to Implementors and press SHIFT-CMD-right arrow).

Try this out: Press SHIFT-Enter and type drawOn: in the Spotter search box. You should see a list showing 5 out of 100 implementors of that method. It shouldn't be all that surprising that so many classes implement this method: drawOn: is the message that is understood by every object that is capable of drawing itself on the screen.

Notice that if you only typed drawOn and left out the colon (:), the number of implementors in the search results is larger. This is because Spotter is doing a partial search, and including any methods that have 'drawOn' in the name, such as drawOn:offset:, drawOnAthensCanvas:, and so on. This is useful for when you want to find a method but can only remember a part of its name.

Method inheritance and overriding

The inheritance browser displays all the methods overridden by the displayed method. To see how it works, select the ImageMorph>>drawOn: method in the browser. Note the arrow icons next to the method name (Figure 1.10). The upward-pointing arrow tells you that ImageMorph>>drawOn: overrides an inherited method (i.e., Morph>>drawOn:), and the downward-pointing arrow tells you that it is overridden by subclasses. (You can also click on the icons to navigate to these methods.) Now right-click on it in the method pane, and select Inheritance. The inheritance browser shows you the hierarchy of overridden methods (see Figure 1.10).

Hierarchy view

By default, the browser presents a list of packages in the leftmost pane. However it is possible to switch to a class hierarchy view. Simply select a particular class of interest, such as ImageMorph and then click on the Hierarchy button (Hier.). You will then see in the second pane a class hierarchy displaying all superclasses and subclasses of the selected class.

Notice that the package pane is disabled, and the packages are greyed out. When you are in Hierarchy view, you cannot change packages. To be able to change them again, toggle out of the Hierarchy view by clicking on the Hierarchy button again.

In Figure 1.11, the hierarchy view reveals that the direct superclass of ImageMorph is Morph.
Some of the key tools of the Pharo environment

Figure 1.10: ImageMorph>>drawOn: and the hierarchy of classes overriding it.

Finding variable references

By right-clicking on a class in the class pane, and selecting Analyze > Inst var references... or Analyze > Class var refs..., you can find out where an instance variable or a class variable is used. You can also have access to those views by clicking on the Variables button, above the package list. Once you click on the button or select the menu item, you will be presented with a dialog that invites you to choose a variable from all of the variables defined in the current class, and all of the variables that it inherits. The list is in inheritance order; it can often be useful to bring up this list just to remind yourself of the name of an instance variable. If you click outside the list, it will go away and no variable browser will be created. If you click on a variable, bounds for example, a Message Browser will be created (Figure 1.12).

You can use a similar method to look at direct variable assignments (that is, places that modify the variable without using accessor methods). Right-click on the class and select Analyze > Inst var assignments.
1.2 The main code browser

[Image: A hierarchy view of ImageMorph]

**Figure 1.11**: A hierarchy view of ImageMorph.

**Bytecode source**

You have the possibility of browsing the bytecode of a method. To do that, right-click on your method and select **Toggle Bytecodes**, or use the shortcut CMD-b CMD-b (see Figure 1.13). Reselect the method again to get back to the normal view.

**Refactorings**

The contextual menus offer a large number of standard refactorings. Simply right-click in any of the four panes to see the currently available refactoring operations. See Figure 1.14.

Refactoring was formerly available only in a special browser called the refactoring browser, but it can now be accessed from any browser.

**Browser menus**

Many additional functions are available by right-clicking in the browser panes. Even if the labels on the menu items are the same, their meaning may be context dependent. For example, the package pane, the class pane, the protocol pane and the method pane all have a **File out** menu item. However, they do different things: the package pane’s **File out** menu item files out the
Some of the key tools of the Pharo environment

![Image of Pharo environment interface]

**Figure 1.12:** A Message Browser for accesses to bounds variable of Morph.

whole package, the class pane’s item files out the whole class, the protocol pane’s item files out the whole protocol, and the method pane’s item files out just the displayed method.

Although this may seem obvious, it can be a source of confusion for beginners. Possibly the most useful menu item is **Find class...** \(\text{(f,c)}\) in the package or class panes. Most of us do not know the package contents of the whole system, and it is much faster to type \text{CMD-f} \text{ CMD-c} followed by the first few characters of the name of a class than to guess which package it might be in.

The **History Navigator** pulldown, found above the protocol and method panes, can also help you quickly go back to a class or method that you have browsed recently, even if you have forgotten its name.

Another useful method in the class pane is **Find method** \(\text{(CMD-f CMD-m)}\), which brings up a menu of all the methods in the class and gives you a search box.

Alternatively, if you are searching for a particular method of the selected class, it is often quicker to browse the **--all--** protocol, place the mouse in the method pane, and type the first letter of the name of the method that you are looking for. This will usually scroll the pane so that the sought-for method
Figure 1.13: Bytecode of the ImageMorph»#DrawOn: compiled method.

Figure 1.14: Refactoring operations.
Some of the key tools of the Pharo environment

name is visible.

To do Try both ways of navigating to OrderedCollection>>removeAt:

There are many other options available in the menus. It pays to spend a few minutes working with the browser and seeing what is there.

Browsing programmatically

The class SystemNavigation provides a number of utility methods that are useful for navigating around the system. Many of the functions offered by the system browser are implemented by SystemNavigation.

Open a playground and evaluate the following code to browse the senders of drawOn::

SystemNavigation default browseAllSendersOf: #drawOn:

To restrict the search for senders to the methods of a specific class:

SystemNavigation default browseAllSendersOf: #drawOn: localTo: ImageMorph

Because the development tools are objects, they are completely accessible from programs and you can develop your own tools or adapt the existing tools to your needs.

The programmatic equivalent to the Implementors of... menu item is:

SystemNavigation default browseAllImplementorsOf: #drawOn:

To do To learn more about what is available, explore the class SystemNavigation with the browser.

1.3 The inspector

One of the things that makes Pharo so different from many other programming environments is that it provides you with a window onto a world of live objects, not a world of static code. Any of those objects can be examined by the programmer, and even modified (although some care is necessary when changing the basic objects that support the system). By all means experiment, but save your image first!

As an illustration of what you can do with an inspector, type DateAndTime now in a playground, and then right-click and choose Inspect it (CMD-i) or Do it and go (CMD-g) (the latter opens an inspector inside the playground window).

Note that it’s often not necessary to select the text before using the menu; if no text is selected, the menu operations work on the whole of the current line.
A window like that shown in Figure 1.15 will appear. This is an inspector, and can be thought of as a window onto the internals of a particular object – in this case, the particular instance of `DateAndTime` that was created when you evaluated the expression `DateAndTime` now. The title bar of the window shows the printable representation of the object that is being inspected.

In the default view (the Raw tab), instance variables can be explored by selecting them in the variable list in the Variable column. As you select a variable, its printable representation is shown in the Value column. More importantly, a separate Inspector view for the selected variable opens in the right hand pane.

For variables that are simple types (booleans, integers, etc), the nested inspector view is not much different from the printable representation in the Value column (although it is a full-fledged Inspector). But for most instance variables, the nested Inspector view on the right has its own Raw tab, with its own list of instance variables. (You can also see that list in the left pane, by expanding the triangle next to a variable’s name.)

You can keep drilling down into the hierarchy of instance variables, with more nested Inspector panes opening to the right of the parent. However, to prevent the multiple panes from being impractical, the panes "scroll" to the right, within the overall inspector window. You can keep track of which "page" you’re on, and also back-track to the original instance that you were inspecting, by using the pagination dots at the bottom of the inspector.
Some of the key tools of the Pharo environment

window.

There are special variants of the inspector for Dictionaries, OrderedCollections, CompiledMethods and a few other classes. These variants have other tabs, in addition to the Raw view, that make it easier to examine the contents of these special objects. For example, an inspector on a Dictionary instance, has an additional Items tab that shows that dictionary’s keys and values in an intuitive fashion.

The horizontal pane at the bottom of the inspector is a small playground window. It is useful because in this window, the pseudo-variable self is bound to the object that you have selected in the left pane. That means you can write and evaluate arbitrary code expressions that use the selected variable’s self, in that bottom pane.

For example, take the inspector on DateAndTime now that you opened earlier in this section. You can select its bottom playground pane, and evaluate the expression self - DateAndTime today. The result will be a Duration object that represents the time interval between midnight today and the instant at which you evaluated DateAndTime now and created the DateAndTime instance that you are inspecting. You can also try evaluating DateAndTime now - self; this will tell you how long you have spent reading this section of this book!

The bottom pane is especially useful if you wanted to change the instance variables of the object being inspected. Provided that you have accessor methods defined for those variables, you can send messages to the root self and change its variables via those accessor methods.

1.4 The debugger

The debugger is arguably the most powerful tool in the Pharo tool suite. It is used not just for debugging, but also for writing new code. To demonstrate the debugger, let’s start by creating a bug!

Using the browser, add the following method to the class String:

```smalltalk
suffix

"assumes that I'm a file name, and answers my suffix, the part after the last dot"

| dot dotPosition |

dot := '.'.

dotPosition := (self size to: 1 by: -1) detect: [ :i | (self at: i) = dot ].

^ self copyFrom: dotPosition to: self size
```

Of course, we are sure that such a trivial method will work, so instead of writing an SUnit test, we just type 'readme.txt' suffix in a playground and
Print it (p). What a surprise! Instead of getting the expected answer 'txt', a PreDebugWindow pops up, as shown in Figure 1.16.

The PreDebugWindow has a title bar that tells us what error occurred, and shows us a stack trace of the messages that led up to the error. Starting from the bottom of the trace, UndefinedObject>>DoIt represents the code that was compiled and run when we selected 'readme.txt' suffix in the playground and asked Pharo to Print it. This code, of course, sent the message suffix to a ByteString object ('readme.txt'). This caused the inherited suffix method in class String to execute; all this information is encoded in the next line of the stack trace, ByteString(String)>>suffix. Working up the stack, we can see that suffix sent detect:... and eventually detect:ifNone sent errorNotFound:.

To find out why the dot was not found, we need the debugger itself, so click on Debug. You can also open the debugger by clicking on any of the lines on the stack trace. If you do this, the debugger will open already focused on the corresponding method.

The debugger is shown in Figure 1.17; it looks intimidating at first, but it is quite easy to use. The title bar and the top pane are very similar to those that we saw in the PreDebugWindow. However, the debugger combines the stack trace with a method browser, so when you select a line in the stack trace, the corresponding method is shown in the pane below. It’s important to realize that the execution that caused the error is still in your image, but in a suspended state. Each line of the stack trace represents a frame on the execution stack that contains all of the information necessary to continue the execution. This includes all of the objects involved in the computation, with
Some of the key tools of the Pharo environment

Figure 1.17: The debugger showing the execution stack and the state of various objects.

their instance variables, and all of the temporary variables of the executing methods.

In Figure 1.17 we have selected the `detect:ifFound:IfNone:` method in the top pane. The method body is displayed in the center pane; the blue highlight around the message `value` shows that the current method has sent the `value` message and is waiting for an answer.

The Variables pane at the bottom of the debugger is actually like a inspector (without playground pane). You can select one variable and open another inspector pane with the well-known inspector panes (Raw, ...), and an Evaluator acting as a Playground for evaluating code within the context of the selected variable. The variables list has up to four types variables it shows.

- **parameter** any parameter passed to this method.
- **temp** any temporaries used in this method.
- **attribute** any instance variable accessible from the context of the current receiver.
- **implicit** pseudo variables (`self`, `thisContext`, `stackTop`) used in the current context.
As you select different stack frames, the identity of self may change, and so will the contents of the Variables list. If you click on self in the bottom-left pane, you will see that self is the interval (10 to: 1 by -1), which is what we expect. You can always select self and select the Evaluator pane to evaluate some code in the content of the current receiver. But because all of the variables are also in scope in the method pane; you should feel free to type or select expressions directly in the method pane and evaluate them. You can always Cancel (1) your changes using the menu or CMD-1.

Selecting thisContext from the list of (implicit) variables, shows the current context object.

As we can see one method lower in the stack trace, the exceptionBlock is [self errorNotFound: ...], so, it is not surprising that we see the corresponding error message.

Incidentally, if you want to open a full inspector on one of the variables shown in the mini-inspectors, just double-click on the name of the variable, or select the name of the variable and right-click to choose Inspect (i). This can be useful if you want to watch how a variable changes while you execute other code.

Looking back at the method window, we see that we expected the penultimate line of the method to find '.' in the string 'readme.txt', and that execution should never have reached the final line. Pharo does not let us run an execution backwards, but it does let us start a method again, which works very well in code that does not mutate objects, but instead creates new ones.

Click Restart, and you will see that the focus of execution returns to the first statement of the current method. The blue highlight shows that the next message to be sent will be do: (see Figure 1.18).

The Into and Over buttons give us two different ways to step through the execution. If you click Over, Pharo executes the current message-send (in this case the do:) in one step, unless there is an error. So Over will take us to the next message-send in the current method, which is value – this is exactly where we started, and not much help. What we need to do is to find out why the do: is not finding the character that we are looking for.

After clicking Over, click Restart to get back to the situation shown in Figure 1.18.

Click Into two times; Pharo will go into the method corresponding to the highlighted message-send, in this case, Interval>>do:.

However, it turns out that this is not much help either; we can be fairly confident that Interval>>do: is not broken. The bug is much more likely to be in what we asked Pharo to do. Through is the appropriate button to use in this case: we want to ignore the details of the do: itself and focus on the execution of the argument block.
Some of the key tools of the Pharo environment

Figure 1.18: The debugger after restarting the `detect:ifFound:IfNone:` method.

Select the `detect:ifFound:IfNone:` method again and Restart to get back to the state shown in Figure 1.18. Now click on Through a few times. Select each in the context window as you do so. You should see each count down from 10 as the do: method executes.

When each is 7 we expect the `ifTrue:` block to be executed, but it isn’t. To see what is going wrong, go Into the execution of `value:` as illustrated in Figure 1.19.

After clicking Into, we find ourselves in the position shown in Figure 1.20. It looks at first that we have gone back to the `suffix` method, but this is because we are now executing the block that `suffix` provided as argument to `detect:`.

If you select dot in the context inspector, you will see that its value is `'. '`. And now you see why they are not equal: the seventh character of `'readme.txt'` is of course a Character, while dot is a String.

Now that we see the bug, the fix is obvious: we have to convert dot to a char-
1.4 The debugger

Figure 1.19: The debugger after stepping Through the do: method several times.

acter before starting to search for it.

Change the code right in the debugger so that the assignment reads dot := $. and accept the change.

Because we are executing code inside a block that is inside a detect:, several stack frames will have to be abandoned in order to make this change. Pharo asks us if this is what we want (see Figure 1.21), and, assuming that we click yes, will save (and compile) the new method.

The evaluation of the expression 'readme.txt' suffix will complete, and print the answer '.txt'.

Is the answer correct? Unfortunately, we can’t say for sure. Should the suffix be .txt or txt? The method comment in suffix is not very precise. The way to avoid this sort of problem is to write an SUnit test that defines the answer.

```smalltalk
testSuffixFound
   self assert: 'readme.txt' suffix = 'txt'
```
Some of the key tools of the Pharo environment

Figure 1.20: The debugger showing why 'readme.txt' at: 7 is not equal to dot.

The effort required to do that was little more than to run the same test in the playground, but using SUnit saves the test as executable documentation, and makes it easy for others to run. Moreover, if you add testSuffix to the class StringTest and run that test suite with SUnit, you can very quickly get back to debugging the error. SUnit opens the debugger on the failing assertion, but you need only go back down the stack one frame, Restart the test and go Into the suffix method, and you can correct the error, as we are doing in Figure 1.22. It is then only a second of work to click on the Run Failures button in the SUnit Test Runner, and confirm that the test now passes.

Here is a better test:

```smalltalk
| dot dotPosition |
dot := ':'.
dotPosition := (self size to: 1 by: -1) detect: [:i | (self at: i) = dot ].
^ self copyFrom: dotPosition to: self size
```

Why is this test better? Because it tells the reader what the method should do if there is more than one dot in the target String.
There are a few other ways to get into the debugger in addition to catching errors and assertion failures. If you execute code that goes into an infinite loop, you can interrupt it and open a debugger on the computation by typing CMD-. (That's a full stop or a period, depending on where you learned English). (It is also useful to know that you can bring up an emergency debugger at any time by typing CMD-SHIFT-.) You can also just edit the suspect code to insert \texttt{Halt now}. So, for example, we might edit the \texttt{suffix} method to read as follows:

```smalltalk
suffix
  "assumes that I'm a file name, and answers my suffix, the part after the last dot"
  | dot dotPosition |
  dotPosition := (self size to: 1 by: -1) detect: [:i | (self at: i) = dot ].
  Halt now.
  ^ self copyFrom: dotPosition to: self size
```

When we run this method, the execution of the \texttt{Halt now} will bring up the pre-debugger, from where we can either proceed, or go into the debugger (and from there look at variables, step through the computation, and edit the code).
Some of the key tools of the Pharo environment

![Image of debugger with code and variables]

**Figure 1.22:** Changing the `suffix` method in the debugger: fixing the off-by-one error after an SUnit assertion failure.

That’s all there is to the debugger, but it’s not all there is to the `suffix` method. The initial bug should have made you realize that if there is no dot in the target string, the `suffix` method will raise an error. This isn’t the behaviour that we want, so let’s add a second test to specify what should happen in this case.

```smalltalk
| dot dotPosition |
dot := $.
dotPosition := (self size to: 1 by: -1) detect: [:i | (self at: i) = dot].
^ self copyFrom: dotPosition + 1 to: self size
```

```smalltalk
testSuffixNotFound
self assert: 'readme' suffix = '
```

Lastly, add `testNoSuffix` to the test suite in class `StringTest`, and watch the test raise an error. Enter the debugger by selecting the erroneous test in SUnit, and edit the code so that the test passes. The easiest and clearest way to do this is to replace the `detect:` message by `detect:ifNone:`, where the second argument is a block that simply returns the string size.

We will learn more about SUnit in Chapter : SUnit.

### 1.5 The process browser

Pharo is a multi-threaded system, and there are many lightweight processes (also known as threads) running concurrently in your image. In the future the
Finding methods

Pharo virtual machine may take advantage of multiple processors when they are available, but at present, concurrency is implemented by time-slicing.

The Process Browser is a cousin of the debugger that lets you look at the various processes running inside Pharo. You can open it using the World Menu, by selecting Tools > Process Browser (figure 1.23 shows a screenshot). The top-left pane lists all of the processes in Pharo, in priority order, from the timer interrupt watcher at priority 80 to the idle process at priority 10. Of course, on a uniprocessor, the only process that can be running when you look is the UI process; all others will be waiting for some kind of event.

By default, the display of processes is static; it can be updated by right-clicking and selecting Turn on auto-update (a).

If you select a process in the top-left pane, its stack trace is displayed in the top-right pane, just as with the debugger. If you select a stack frame, the corresponding method is displayed in the bottom pane. The process browser is not equipped with mini-inspectors for self and thisContext, but right-clicking on the stack frames provide equivalent functionality.

1.6 Finding methods

The Finder is one of several code search tools in Pharo to help you find methods by name (or even functionality). We’ve discussed it in some length in
Chapter: A Quick Tour of Pharo.

1.7 Chapter summary

In order to develop effectively with Pharo, it is important to invest some effort into learning the tools available in the environment.

- The standard browser is your main interface for browsing existing packages, classes, method protocols and methods, and for defining new ones.

- The browser offers several useful shortcuts to directly jump to senders or implementors of a message, versions of a method, and so on.

- From any of the tools, you can highlight the name of a class or a method and immediately jump to a browser by using the keyboard shortcut CMD-b.

- You can also browse the Pharo system programmatically by sending messages to SystemNavigation default.

- The Inspector is a tool that is useful for exploring and interacting with live objects in your image.

- You can even inspect tools by meta-clicking to bring up their morphic halo and selecting the debug handle.

- The Debugger is a tool that not only lets you inspect the run-time stack of your program when an error is raised, but it also enables you to interact with all of the objects of your application, including the source code. In many cases you can modify your source code from the debugger and continue executing. The debugger is especially effective as a tool to support test-first development in tandem with SUnit (Chapter: SUnit).

- The Process Browser lets you monitor, query and interact with the processes current running in your image.

- The Finder is a tool for locating methods.