Pharo by Example 8.0

Stéphane Ducasse, Sebastijan Kaplar, Gordana Rakic and Quentin Ducasse

March 8, 2021

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Preface

This version of the book is based on the previous version authored by: Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker called, *Pharo by Example*. It is also built on the version that was issued for Pharo 50 named *Updated Pharo by Example* authored by: Stéphane Ducasse and edited by Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. The current version got many edits, modifications and updates to bring the book to the current version of Pharo.

Maintaining a book to be update to date is not an easy task. This is not just changing a 5.0 into a 8.0. This is a tedious and often boring work. Many aspects of Pharo from a tooling perspective have changed such as the git support with Iceberg.

- We cover briefly the new system browser, Calypso.
- This book version has two new chapters covering Iceberg and package management (see Chapters 5 and 6) - a new book on Pharo and git is available at http://books.pharo.org.
- We simplified the SUnit chapter, since a companion book is available now at http://books.pharo.org/
- Altogether we revise parts of the books that were wrong such as Morphic chapter, in previous editions and the current version has really improved over Pharo by Example 5.

Pharo by Example is not magically updating itself but the result of work. This is why I decided, as one of the original and main author over the years, to invite Sebastijan, Gordana and Quentin to be listed as authors of Pharo by Example8.0.

1.1 What is Pharo?

Pharo is a modern, open-source, dynamically-typed language supporting live coding inspired by Smalltalk. Pharo and its ecosystems are composed of six fundamental elements:

- A dynamically-typed language with a syntax so simple it can fit on a postcard and yet is readable even for someone unfamiliar with it.
- A live coding environment that allows the programmer to modify code while the code executes, without any need to slow down their workflow.
- A powerful IDE providing all the tools to help manage complex code and promote good code design.
- A rich library that creates an environment so powerful that it can be viewed even as a virtual OS, including a very fast JITing VM and full access to OS libraries and features via its FFI.
- A culture where changes and improvements are encouraged and highly valued.
- A community that welcomes coders from any corner of the world with any skill and any programming languages.

Pharo strives to offer a lean, open platform for professional software development, as well as a robust and stable platform for research and development into dynamic languages and environments. Pharo serves as the reference implementation for the Seaside web development framework available at http://www.seaside.st.

Pharo core contains only code that has been contributed under the MIT license. The Pharo project started in March 2008 as a fork of Squeak (a modern implementation of Smalltalk-80), and the first 1.0 beta version was released on July 31, 2009. Since then Pharo got a new version each year or year and an half. The current version is Pharo 8.0, released in January 2020.

Pharo is highly portable. Pharo can run on OS X, Windows, Linux, Android, iOS, and Raspberry Pi. Its virtual machine is written entirely in a subset of Pharo itself, making it easy to simulate, debug, analyze, and change from within Pharo itself. Pharo is the vehicle for a wide range of innovative projects, from multimedia applications and educational platforms to commercial web development environments.

There is an important principle behind Pharo: Pharo does not just copy the past, it reinvents the essence of Smalltalk. However we realize that Big Bang style approaches rarely succeed. Pharo instead favors evolutionary and incremental changes. Rather than leaping for the final perfect solution in one big step, a multitude of small changes keeps even the bleeding edge relatively stable while experimenting with important new features and libraries.

This facilitates contributions and rapid feedback from the community, on which Pharo relies on for its success. Finally Pharo is not read-only, Pharo integrates changes made by the community, daily. Pharo has around 100 contributors, based all over the world. You can have an impact on Pharo too! Check http://github.com/pharo-project/pharo.

1.2 Who should read this book?

The previous revision of this book was based on Pharo 5.0. This revision has been liberally updated to align with Pharo 8.0. Various aspects of Pharo are presented, starting with the basics then proceeding to intermediate topics.

An excellent MOOC (Massive online course) is freely available for Pharo at http://mooc.pharo.org.

This book will not teach you how to program. The reader should have some familiarity with programming languages. Some background with object-oriented programming would also be helpful.

The current book will introduce the Pharo programming environment, the language and the associated tools. You will be exposed to common idioms and practices, but the focus is on the technology, not on object-oriented design. Wherever possible, we will show you lots of examples.

There are numerous other books on Smalltalk freely available on the web at http://stephane.ducasse.free.fr/FreeBooks.html.

Further readings

This book is not alone. Here is a litle commented list of possible other readings that you can find at http://books.pharo.org.

- ”Learning Object-Oriented Programming, Design and TDD with Pharo”. This book is teaching key aspects of object design and test driven development. If you are learning

Object-oriented programming this is a good for you.

- ”Pharo with Style”. This book is a must read. It discusses how to write good and readable Pharo code. In one hour, you will boost your coding standard.
- ”The Spec UI framework”. This book will show you how to develop user interface standard applications in Pharo.

More technical books are:

- ”Managing your code with Iceberg”. This book covers with a bit more depth how to manage your code with git.

- ”Enterprise Pharo”. This book contains different chapters related to web, converters, reporting documents that you need for delivering applications.
- ”Deep into Pharo”. This book covers more advanced topics than Pharo by Example.

Then you have books to expand your mind:

- ”A simple reflective object kernel” revisits all the fundamental points of ”objects all the way down” by driving you in a little journey to build

a little reflective language core. It is truly excellent.

1.3 A word of advice

Do not be frustrated by parts of Pharo that you do not immediately understand. You do not have to know everything! Alan Knight expresses this as follows:

Try not to care. Beginning Pharo programmers often have trouble because they think they need to understand all the details of how a thing works before they can use it. This means it takes quite a while before they can master Transcript show: 'Hello World'. One of the great leaps in OO is to be able to answer the question “How does this work?” with ”I don’t care”.

When you do not understand something, simple or complex, do not hesitate for a second to ask us at our mailing lists (pharo-users@lists.pharo.org or pharo-dev@lists.pharo.org), irc and Discord. We love questions and we welcome people of any skill.

1.4 An open book

This book is an open book in the following senses:

- The content of this book is released under the Creative Commons Attribution-ShareAlike (by-sa) license. In short, you are allowed to freely share and adapt this book, as long as you respect the conditions of the license available at the following URL http://creativecommons.org/licenses/by-sa/3.0/.
- This book just describes the core of Pharo. We encourage others to contribute chapters on the parts of Pharo that we have not described. If you would like to participate in this effort, please contact us. We would like to see more books around Pharo!
- It is also possible to contribute directly to this book via Github. Just follow the instructions there and ask any question on the mailing list. You can find the Github repo at https://github.com/SquareBracketAssociates/PharoByExample80

Listing 1-1 Small example

```
3 + 4
>>> 7 "if you select 3+4 and 'print it', you will see 7"
```

1.5 The Pharo community

The Pharo community is friendly and active. Here is a short list of resources that you may find useful:

- http://www.pharo.org is the main web site of Pharo.
- http://www.github.com/pharo-project/pharo is the main github account for Pharo.
- Pharo has an active on Discord server - a platform for chat based on IRC, just ask for an invitation on Pharo's website http://pharo.org/community, in the discord section. Everybody is welcomed.
- Pharoers started a wiki on Pharo https://github.com/pharo-open-documentation/pharo-wiki
- An Awesome catalog is maintained with projects: https://github.com/pharo-open-documentation/awesome-pharo
- If you hear about SmalltalkHub, http://www.smalltalkhub.com/ was the equivalent of SourceForge/Github for Pharo projects for about 10 years. Many extra packages and projects for Pharo lived there. Now the community is mainly using git repositories such as github.com, gitlab and bitbucket.

1.6 Examples and exercises

We have tried to provide as many examples as possible. In particular, there are many examples that show a fragment of code which can be evaluated. We use a long arrow to indicate the result you obtain when you select an expression and from its context menu choose **print it**:

In case you want to play with these code snippets in Pharo, you can download a plain text file with all the example code from the Resources sidebar of the original book's web site: http://books.pharo.org/pharo-by-example.

1.7 Acknowledgments

We would like to thank Alan Kay, Dan Ingalls and their team for making Squeak, an amazing Smalltalk development environment, that became the open-source project from which Pharo took roots. Pharo also would not be possible without the incredible work of the Squeak developers.

We would also like to thank Hilaire Fernandes and Serge Stinckwich who allowed us to translate parts of their columns on Smalltalk, and Damien Cassou for contributing the chapter on Streams. We especially thank Alexandre Bergel, Orla Greevy, Fabrizio Perin, Lukas Renggli, Jorge Ressia and Erwann Wernli for their detailed reviews. We thank the University of Bern, Switzerland, for graciously supporting this open-source project and for hosting the web site of this book during some years.

We also thank the Pharo community for their enthusiastic support of this book project, as well as for all the translations of the first edition of **Pharo by Example**.

1.8 Hyper special acknowledgments

We want to thank the original authors of this book! Without this initial version it would have been difficult to make this one. Pharo by Example is a central book to welcome newcomers and it has a great value.

Thanks to Manfred Kröhnert, Markus Schlager, Werner Kassens, Michael OKeefe, Aryeh Hoffman, Paul MacIntosh, Gaurav Singh, Jigyasa Grover, Craig Allen, Serge Stinckwich, avh-on1, Yuriy Timchuk, zio-pietro for the typos and feedback. Special thanks to Damien Cassou and Cyril Ferlicot for their great help in the book update.

Finally we want to thank Inria for its steady and important financial support, and the RMoD team members for the constant energy pushing Pharo forward.

Super special thanks to Damien Pollet for this great book template.

S. Ducasse, S. Kaplar, Gordana Rakic, and Q. Ducasse

CHAPTER 2

A quick tour of Pharo (to revisit - started by Gordana)

This chapter will take you on a high level tour of Pharo, to help you get comfortable with the environment. There will be plenty of opportunities to try things out, so it would be a good idea if you have a computer handy when you read this chapter.

In particular, you will fire up Pharo, learn about the different ways of interacting with the system, and discover some of the basic tools. You will also learn how to define a new method, create an object and send it messages.

Note: Most of the introductory material in this book will work with any Pharo version, so if you already have one installed, you may as well continue to use it. However, since this book is written for Pharo 8, if you notice differences between the appearance or behaviour of your system and what is described here, do not be surprised.

2.1 Installing Pharo

Pharo does not need to install anything in your system, as it's perfectly capable of running standalone. As it will be explained later, Pharo consists of the virtual machine (VM), the image, the changes and the sources.

Depending on your platform, download the appropriate `.zip` file, uncompress it in a directory of your choice and now you are ready to launch Pharo. Pharo can be also installed via the command line.

Downloading Pharo

Pharo is available as a free download from http://pharo.org/download. You can download either Pharo Launcher some of the Pharo standalone versions.

Pharo Launcher is a cross-platform application that allows easy managing of Pharo images. Click the button for your operating system to download the appropriate Pharo Launcher, it contains everything you need to run Pharo.

For standalone version download the appropriate `.zip` file. Once the file is unzipped, it will contain everything you need to run Pharo.

Using handy scripts

http://files.pharo.org/get/ offers a collection of scripts to download specific versions of Pharo. This is really handy to automate the process.

To download the latest 8.0 full system, use the following snippet.

```
wget -O- get.pharo.org/80+vm | bash
```

Then you can execute the following

```
./pharo-ui Pharo.image
```

2.2 Pharo: File Components

Pharo consists of four main component files. Although you do not need to deal with them directly for the purposes of this book, it is useful to understand the roles they play.

1. The **virtual machine** (VM) is the only component that is different for each operating system. The VM is the execution engine (similar to a JVM). It takes Pharo bytecode that is generated each time user compiles a piece of code, converts it to machine code and executes it. Pharo comes with the Cog VM a very fast JITing VM. The VM executable is named:

- `Pharo.exe` for Windows;
- `pharo` for Linux ; and
- `Pharo` for OSX (inside a package also named `Pharo.app`).

The other components below are portable across operating systems, and can be copied and run on any appropriate virtual machine.

2. The **sources** file contains source code for parts of Pharo that do not change frequently. Sources file is important because the image file format stores only objects including compiled methods and their bytecode and not their source code. Typically a new **sources** file is generated once per major release of Pharo. For Pharo 8.0, this file is named `PharoV80.sources`.

3. The **changes** file logs of all source code modifications (especially all the changes you did while programming) since the `.sources` file was generated. Each release provides a near empty file named for the release, for example `Pharo8.0.changes`. This facilitates a per method history for diffs or reverting. It means that even if you did not manage to save the image file on a crash or you just forgot, you can recover your changes from this file. A changes file is always coupled with a image file. They work in pair.

4. The **image** file provides a frozen in time snapshot of a running Pharo system. This is the file where all objects are stored and as such it's a cross platform format. An image file contains the live state of all objects of the system at a given point, including classes and compiled methods since they are objects, too. An image is a virtual object container. The file is named for the release (like `Pharo8.0.image`) and it is synched with the `Pharo8.0.changes` file.

Image/Changes Pair

The `.image` and `.changes` files provided by a Pharo release are the starting point for a live environment that you adapt to your needs. As you work in Pharo, these files are modified, so you need to make sure that they are writable. Pay attention to remove the changes and image files from the list of files to be checked by anti-viruses. The `.image` and `.changes` files are intimately linked and should always be kept together, with matching base filenames. Never edit them directly with a text editor, as `.images` holds your live object runtime memory, which indexes into the `.changes` files for the source. It is a good idea to keep a backup copy of the downloaded `.image` and `.changes` files so you can always start from a fresh image and reload your code. However, the most efficient way for backing up code is to use a version control system that will provide an easier and powerful way to back up and track your changes.

Common Setup

The four main component files above can be placed in the same directory, but it's a common practice to put the Virtual Machine and sources file in a separate directory where everyone has read-only access to them.

Do whatever works best for your style of working and your operating system.

2.3 Launching Pharo

To start Pharo, if you are using Pharo Launcher, select the image you wish to use and press launch. Or if you are using standalone version do whatever your operating system expects: drag the .image file onto the icon of the vir-

Listing 2-1 Launching pattern

```
<Pharo executable> <path to Pharo image>
```

Listing 2-2 Launching Pharo from Linux

```
./pharo shared/Pharo8.0.image
```

Listing 2-3 Launching Pharo from Mac OS X

```
Pharo8.0.app/Contents/MacOS/Pharo
    Pharo8.0.app/Contents/Resources/Pharo8.0.image
```

tual machine, or double-click the .image file, or at the command line type the name of the virtual machine followed by the path to the .image file.

- On **OS X**, double click the `Pharo8.0.app` bundle in the unzipped download.
- On **Linux**, double click (or invoke from the command line) the `pharo` executable `Bash` script from the unzipped Pharo folder.
- On **Windows**, enter the unzipped Pharo folder and double click `Pharo.exe`.

In general, Pharo tries to "do the right thing". If you double click on the VM, it looks for an image file in the default location. If you double click on an `.image` file, it tries to find the nearest VM to launch it with.

If you have multiple VMs installed on your machine, the operating system may no longer be able to guess the right one. In this case, it is safer to specify exactly which ones you meant to launch, either by dragging and dropping the image file onto the VM, or specifying the image on the command line (see the next section).

2.4 Launching Pharo Via the Command Line

The general pattern for launching Pharo from a terminal is:

Linux command line.

For Linux, assuming that you're in the unzipped `pharo8.0` folder:

OS X command line.

For OS X, assuming that you're in the directory with the unzipped `Pharo8.0.app` bundle:

When using a Pharo bundle, you need to right-click on `Pharo8.0.app` and select 'Show Package Contents' to get access to the image. If you need this often, just download a separated image/changes pair and drop that image into the `Pharo8.0.app`.

Listing 2-4 Launching Pharo from Windows

```
Pharo.exe Pharo8.0.image
```

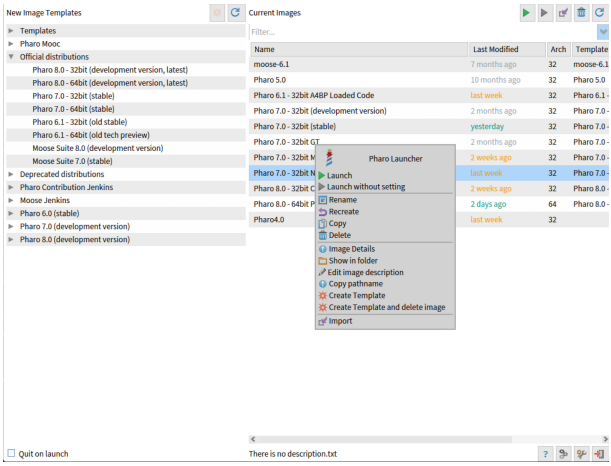

Figure 2-5 PharoLauncher - GUI.

Windows command line.

For Windows, assuming that you're in the unzipped `Pharo8.0` folder:

2.5 Pharo Launcher

PharoLauncher is a tool that helps you download and manage Pharo images. It is very useful for getting new versions of Pharo (as well as updates to the existing versions that contain important bug fixes). It also gives you access to images preloaded with specific libraries that make it very easy to use those tools without having to manually install and configure them.

PharoLauncher can be found on GitHub at https://github.com/pharo-project/pharo-launcher.[1] together with installation instructions and download links depending on your platform. **PharoLauncher** is basically composed of two columns.

After installing PharoLauncher and opening it (like you would do for any Pharo image), you should get a GUI similar to Figure 2-5.

The right column lists images that live locally on your machine (usually in a shared system folder). You can launch any local image directly (either by double-clicking, or by selecting it and pressing the `Launch` button). A right-

[1] https://github.com/pharo-project/pharo-launcher

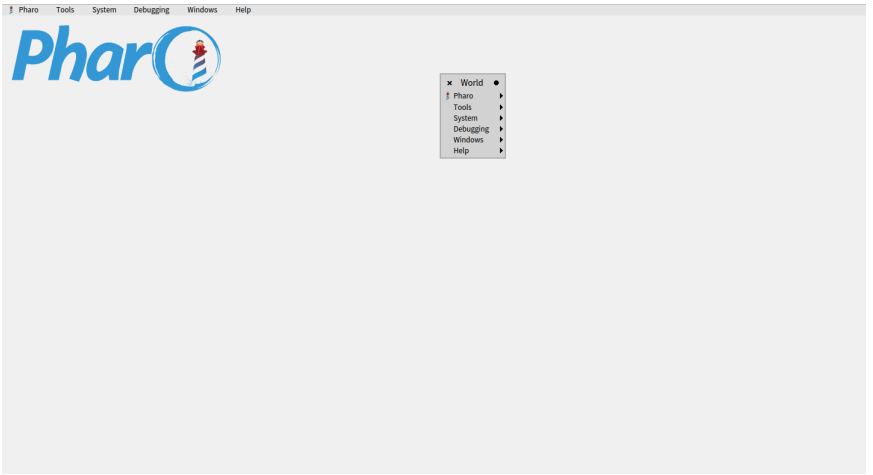

Figure 2-6 Clicking anywhere on the Pharo window background activates the World Menu.

click context menu provides several useful functions like copying and renaming your images, as well as locating them on the file system.

The left column lists Templates, which are remote images available for download. To download a remote image, select it and click the `Create image` button (located on the top right, next to the `Refresh template list` button).

You can use your own local images with **PharoLauncher**, in addition to working with the images you downloaded. To do so, simply ensure that your .image and its associated .changes files are placed in a folder (with the same name as your image) in your default image location. You can find the location in the **PharoLauncher** settings.

2.6 The World Menu

Once Pharo is running, you should see a single large window, possibly containing some open playground windows (see Figure 2-6). You might notice a menu bar, but Pharo mainly makes use of context-dependent pop-up menus.

Clicking anywhere on the background of the Pharo window will display the **World Menu**, which contains many of the Pharo tools, utilities and settings.

At the top of the **World Menu**, in the **Tools** submenu you will see a list of several core tools in Pharo, including the System Browser, the Playground, the Monticello package manager, and others. We will discuss them in more detail in the coming chapters.

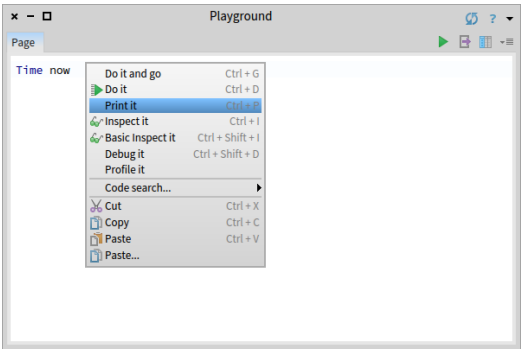

Figure 2-7 Action Click (right click) brings the contextual menu.

2.7 Interacting with Pharo

Pharo offers three ways to interact with the system using a mouse or other pointing device.

click (or left-click): this is the most often used mouse button, and is normally equivalent to left-clicking (or clicking a single-mouse button without any modifier key). For example, click on the background of the Pharo window to bring up the `World` menu (Figure 2-6).

action-click (or right-click): this is the next most used button. It is used to bring up a contextual menu that offers different sets of actions depending on where the mouse is pointing (see Figure 2-7). If you do not have a multi-button mouse, then normally you will configure the control modifier key to action-click with the mouse button.

meta-click: Finally, you may meta-click on any object displayed in the image to activate the ”morphic halo”, an array of handles that are used to perform operations on the on-screen objects themselves, such as inspecting or resizing them (see Figure 2-8). If you let the mouse linger over a handle, a help balloon will explain its function. In Pharo, how you meta-click depends on your operating system: either you must hold `Shift-Ctrl` or `Shift-Alt` (on Windows or Linux) or `Shift-Option` (on OS X) while clicking.

2.8 Sending Messages

In the Pharo window, click on an open space to open the **World Menu**, and then in the **Tools** submenu select the **Playground** menu option. The **Playground** tool will open (you may recognize it as the **Workspace** tool, from previous versions of Pharo). We can use **Playground** to quickly execute Pharo code. Enter the following code in it, then right click and select `Do it`:

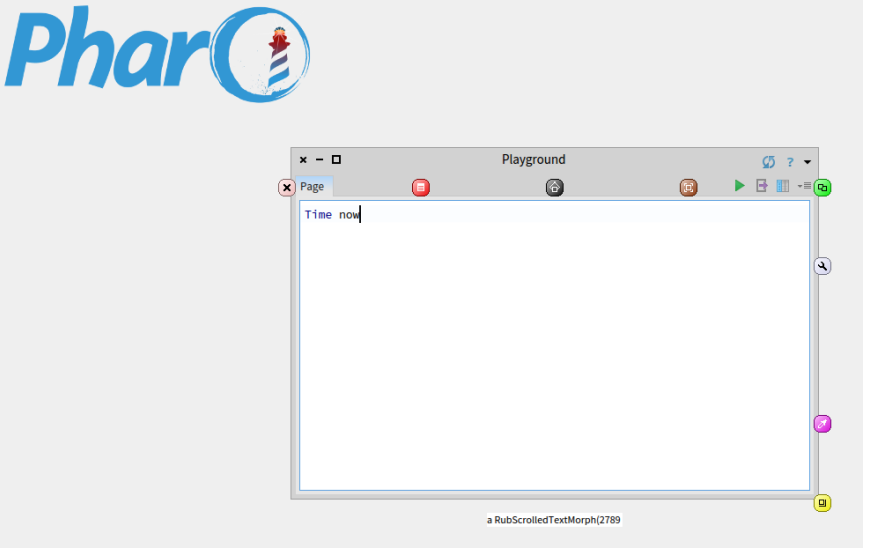

Figure 2-8 Meta-Clicking on a window opens the Halos.

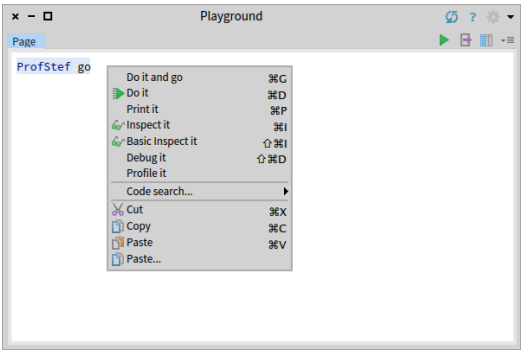

Figure 2-10 Executing an expression is simple with the `Do it` menu item.

Listing 2-9 Open ProfStef in the `Playground`.

```
ProfStef go.
```

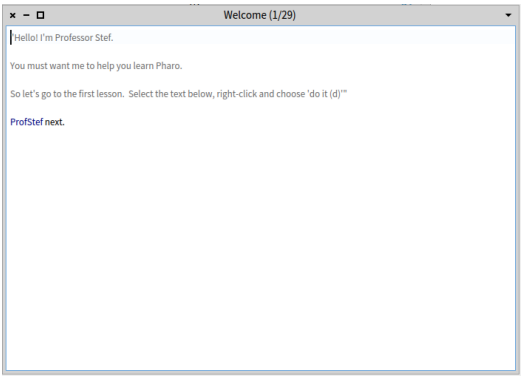

Figure 2-11 PharoTutorial is a simple interactive tutorial to learn about Pharo.

This expression will trigger the Pharo tutorial (as shown in Figure 2-11). It is a simple and interactive tutorial that will teach you the basics of Pharo.

Congratulations, you have just sent your first message! Pharo is based on the concept of sending messages to objects. The Pharo objects are like your soldiers ready to obey once you send them a message they can understand. We will see how an object can understand a message, later on.

If you talk to Pharoers for a while, you will notice that they generally do not use expressions like *call an operation* or *invoke a method*, as developers do in other programming languages. Instead they will say *send a message*. This reflects the idea that objects are responsible for their own actions and that the method associated with the message is looked up dynamically. When sending a message to an object, the object, and not the sender, selects the appropriate method for responding to your message. In most cases, the method with the same name as the message is executed.

As a user you do not need to understand how each message works, the only thing you need to know is what the available messages are for the objects that interest you. This way an object can hide its complexity, and coding can be kept as simple as possible without losing flexibility.

How to find the available messages for each object is something we will explore later on.

2.9 Saving, Quitting and Restarting a Pharo Session

You can exit Pharo at any point, by closing the Pharo window as you do any other application window. Additionally you can use the **World Menu** and

select either `Save and quit` or `Quit` in the **Pharo** submenu.

In any case, Pharo will display a prompt to ask you about saving your image. If you do save your image and reopen it, you will see that things are *exactly* as you left them. This happens because the image file stores all the objects (edited text, window positions, added methods... of course because they are all objects) that Pharo has loaded into your memory so that nothing is lost on exit.

When you start Pharo for the first time, the Pharo virtual machine loads the image file that you specified. This file contains a snapshot of a large number of objects, including a vast amount of pre-existing code and programming tools (all of which are objects). As you work with Pharo, you will send messages to these objects, you will create new objects, and some of these objects will die and their memory will be reclaimed (garbage-collected).

When you quit Pharo, you will normally save a snapshot that contains all of your objects. If you save normally, you will overwrite your old image file with the new snapshot. Alternatively, you may save the image under a new name.

As mentioned earlier, in addition to the `.image` file, there is also a `.changes` file. This file contains a log of all the changes to the source code that you have made using the standard tools. Most of the time you do not need to worry about this file at all. As we shall see, however, the `.changes` file can be very useful for recovering from errors, or replaying lost changes. More about this later!

It may seem like the image is the key mechanism for storing and managing software projects, but that is not the case. As we shall see soon, there are much better tools for managing code and sharing software developed by teams. Images are very useful, but you should learn to be very cavalier about creating and throwing away images, since versioning tools like `Monticello` and `Iceberg` offer much better ways to manage versions and share code amongst developers. In addition, if you need to persist objects, you can use several systems such as `Fuel` (a fast object binary serializer), `STON` (a textual object serializer) or a database.

2.10 Playgrounds and Transcripts

Let us start with some exercises:

1. Close all open windows within Pharo.
2. Open a Transcript and a Playground/workspace. (The Transcript can be opened from the `World > Tools > ...` submenu.)
3. Position and resize the transcript and playground windows so that the playground just overlaps the transcript (see Figure 2-12).

You can resize windows by dragging one of the corners. At any time only one window is active; it is in front and has its border highlighted.

About Transcript.

The `Transcript` is an object that is often used for logging system messages. It is a kind of *system console*.

About Playground.

`Playgrounds` are useful for typing snippets of code that you would like to experiment with. You can also use playgrounds simply for typing any text that you would like to remember, such as to-do lists or instructions for anyone who will use your image.

Type the following text into the playground:

```
Transcript show: 'hello world'; cr.
```

Try double-clicking at various points on the text you have just typed. Notice how an entire word, entire string, or all of the text is selected, depending on whether you click within a word, at the end of the string, or at the end of the entire expression. In particular, if you place the cursor before the first character or after the last character and double-click, you select the complete paragraph.

Select the text you have typed, right click and select `Do it`. Notice how the text "hello world" appears in the transcript window (See Figure 2-12). Do it again.

2.11 Keyboard Shortcuts

If you want to evaluate an expression, you do not always have to right click. Instead, you can use keyboard shortcuts shown in menu items. Even though Pharo may seem like a mouse driven enviroment it contains over 200 shortcuts that allow you operate a variaty of tools, as well as the facility to assign a keyboard shortcut to any of the 110000 methods contained in the Pharo image. To have a look at the available shortcuts go to World Menu > System > Keymap Browser.

Depending on your platform, you may have to press one of the modifier keys which are Control, Alt, and Command. We will use `CMD` in the rest of the book: so each time you see something like `CMD-d`, just replace it with the appropriate modifier key depending on your OS. The corresponding modifier key in Windows is `CTRL`, and in Linux is either `ALT` or `CTRL`, so each time you see something like `CMD-d`, just replace it with the appropriate modifier key depending on your OS.

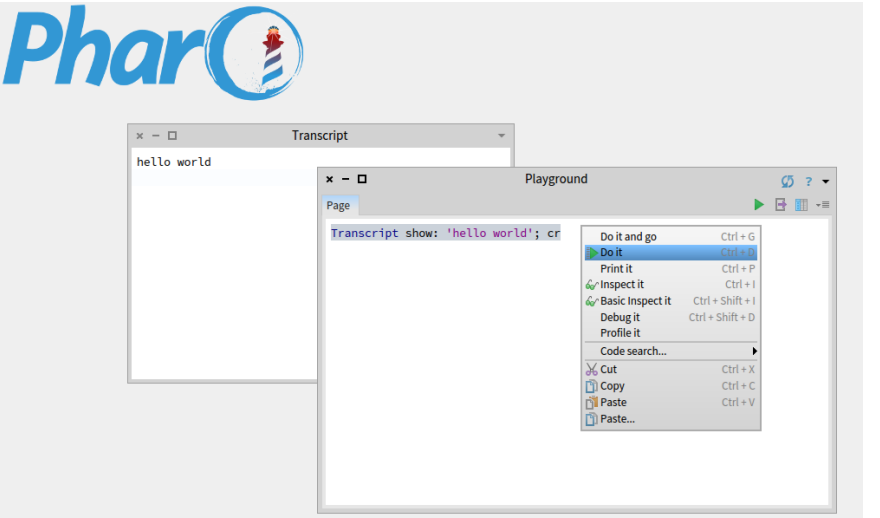

Figure 2-12 Executing an expresssion: displaying a string in the Transcript.

In addition to `Do it`, you might have noticed `Do it and go`, `Print it`, `Inspect it` and several other options in the context menu. Let's have a quick look at each of these.

2.12 Doing vs. Printing

Type the expression `3 + 4` into the playground. Now `Do it` with the keyboard shortcut.

Do not be surprised if you saw nothing happen! What you just did is send the message `+` with argument `4` to the number `3`. Normally the resulting `7` would have been computed and returned to you, but since the playground did not know what to do with this answer, it simply did not show the answer. If you want to see the result, you should `Print it` instead. `Print it` actually compiles the expression, executes it, sends the message `printString` to the result, and displays the resulting string.

Select `3+4` and `Print it` (`CMD-p`). This time we see the result we expect.

```
3 + 4
>>> 7
```

We use the notation >>> as a convention in this book to indicate that a particular Pharo expression yields a given result when you `Print it`.

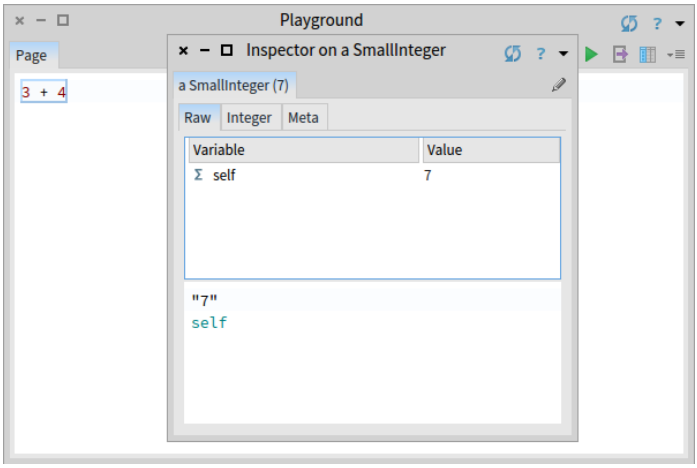

Figure 2-13 Inspecting a simple number using `Inspect`.

2.13 Inspect

Select or place the cursor on the line of `3+4`, and this time `Inspect it` (`CMD-i`).

Now you should see a new window titled "Inspector on a SmallInteger(7)" as shown in Figure 2-13. The inspector is an extremely useful tool that allows you to browse and interact with any object in the system. The title tells us that 7 is an instance of the class `SmallInteger`. The top panel allows us to browse the instance variables of an object and their values. The bottom panel can be used to write expressions to send messages to the object. Type `self squared` in the bottom panel of the inspector, and `Print it`.

The inspector presents specific tabs that will show different information and views on the object depending on the kind of object you are inspecting. Inspect `Morph new openInWorld` you should get a situation similar to the one of Figure 2-14.

2.14 Other Operations

Other right-click options that may be used are the following:

- `Do it and go` additionally opens a *navigable* inspector on the side of the playground. It allows us to navigate the object structure. Try with the previous expression `Morph new openInWorld` and navigate the structure.
- `Basic Inspect it` opens the classic inspector that offers a more minimal GUI and live updates of changes to the object.

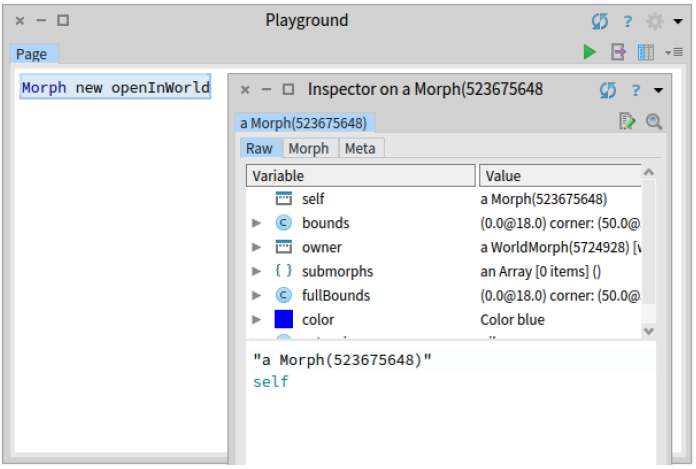

Figure 2-14 Inspecting a Morph using `Inspect`.

- `Debug it` opens the debugger on the code.
- `Profile it` profiles the code with the Pharo profile tool which shows how much time is spent for each message sent.
- `Code search` offers several options provided by System Browser, such as browsing the source code of an expression, searching for senders and implementors, and so on.

2.15 The System Browser

The `System Browser`, also known as "Class Browser", is one of the key tools used for programming. As we shall see, there are several interesting browsers available for Pharo, but this is the basic one you will find in any image. The current implementation of the System Browser is called Calypso. Previous version of the System Browser was called Nautilus.

2.16 Opening the System Browser on a Given Method

This is not the usual way that we open a browser on a method: we use more advanced tools! But for the sake of this exercise, execute the following code snippet:

```
ClyFullBrowser openOnMethod: Integer>>#factorial
```

It will open a system browser on the method `factorial`. We should get a System Browser like in Figure 2-15. The title bar indicates that we are browsing the class `Integer` and its method `factorial`. Figure 2-15 shows the dif-

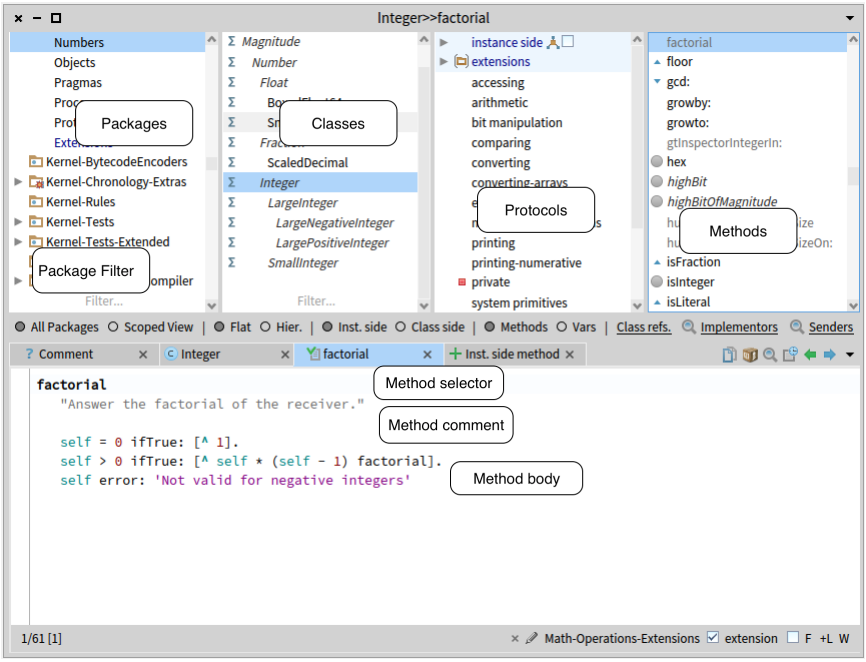

Figure 2-15 The System Browser showing the `factorial` method of class `Integer`.

ferent entities displayed by the browser: packages, classes, protocols, methods and method definition.

In Pharo, the default System Browser is Calypso. However, it is possible to have other System Browsers installed in the Pharo enviroment such as AltBrowser. Each System Browser may have its own GUI that may be very different from the Calypso GUI. From now on, we will use the terms `Browser`, `System Browser` and `Calypso` interchangeably.

2.17 Navigating Using the System Browser

Pharo has **Spotter** (see below) to navigate the system. Now we just want to show you the working flow of the System Browser. Usually with **Spotter** we go directly to the class or the method.

Let us look how to find the `printString` method defined in class `Object`. At the end of the navigation, we will get the situation depicted in 2-16.

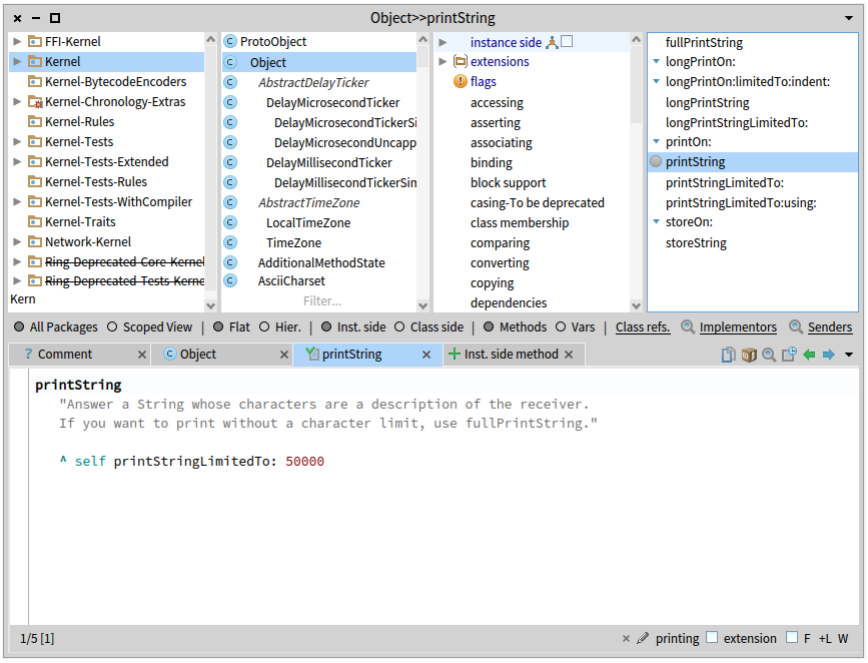

Figure 2-16 The System Browser showing the `printString` method of class `Object`.

Open the Browser by selecting `World > Tools > System Browser`.

When a new System Browser window first opens, all panes but the leftmost are empty. This first pane lists all known packages, which contain groups of related classes.

Filter packages.

Type part of the name of the package in the left most filter. It filters the list of packages to be shown in the list above it. Type 'Kern' for example.

Expand the `Kernel` package and select the `Object` element.

When we select a package, it causes the second pane to show a list of all of the classes in the selected package. You should see the hierarchy of `ProtoObject`.

Select the `Object` class.

When a class is selected, the remaining two panes will be populated. The third pane displays the protocols of the currently selected class. These are

convenient groupings of related methods. If no protocol is selected you should see all methods in the fourth pane.

Select the `printing` protocol.

You may have to scroll down to find it. You can also click on the third pane and type `pr`, to typeahead-find the `printing` protocol. Now select it, and you will see in the fourth pane only methods related to printing.

Select the `printString` Method.

Now we see in the bottom pane the source code of the `printString` method, shared by all objects in the system (except those that override it).

There are much better way to find a method and we will look at them now.

2.18 Finding Classes

There are several ways to find a class in Pharo. The first, as we have just seen above, is to know (or guess) what package it is in, and to navigate to it using the browser.

A second way is to send the `browse` message to the class, asking it to open a browser on itself. Suppose we want to browse the class `Point`.

2.19 Using the Message `browse`

Type `Point browse` into a playground and `Do it`. A browser will open on the `Point` class.

2.20 Using `CMD-b` to Browse

There is also a keyboard shortcut `CMD-b` (browse) that you can use in any text pane; select the word and press `CMD-b`. Use this keyboard shortcut to browse the class `Point`.

Notice that when the `Point` class is selected but no protocol or method is selected, instead of the source code of a method, we see a class definition. This is nothing more than an ordinary message that is sent to the parent class, asking it to create a subclass. Here we see that the class `Object` is being asked to create a subclass named `Point` with two instance variables, class variables, and to put the class `Point` in the `Kernel-BasicObjects` package. If you click on the `Comments` button at the bottom of the class pane, you can see the class comment in a dedicated pane.

In addition the system supports the following mouse shortcuts

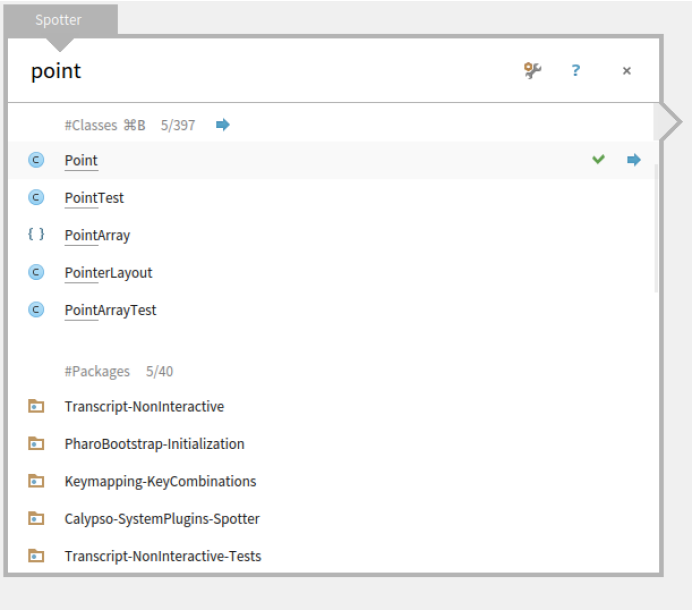

Figure 2-17 Opening Spotter.

- `CMD-Click` on a word: open the definition of a class when the word is a class name. You get also the implementors of the message when you click on a selector that is in the body of a method.
- `CMD-Shift-Click` on a word: open a list browser with all the refs of the class when the word is a class name. You get also the senders of the message when you click on a selector that is in the body of a method.

2.21 Using Spotter

The fastest (and probably the coolest) way to find a class is to use **Spotter**. Pressing `Shift+Enter` opens `Spotter`, a very powerful tool for finding classes, methods, and many other related actions. Figure 2-17 shows that we look for `Point`.

Spotter offers several possibilities as shown in Figure 2-17. You can specify to Spotter the kind of *categories* you are interested in. For example, using `#class` followed by the word you look for, indicates that you are interested in classes. This is the default so you do not need to type `#class`.

Figure 2-18 shows how we can ask **Spotter** to show all the implementors of a given messages. We do not have to type the full category name. Other Categories are menu, packages, method (`#implementor`), examples (`#example`), pragma (`#pragma`), senders (`#sender`), class references (`#reference`) but

Figure 2-18 Looking for implementors matching `printString`.

also playground code snippets (using `#playground`).You can just type the beginning of the category to identify it i.e., `#ref Point` will give all the reference to the class `Point`.

Spotter can be used even to browse through the OS file system, and has a history category where previous searches are stored for quickly going back to popular searches.

Navigating Results

In addition we can use **Spotter** to navigate to our search results similarly to how we use System Browser. Spotter categorizes its search results: for example, classes are under Classes category, methods under the Implementors category, help topics under Help Topics category, etc.

Clicking on the right arrow will take us to our selection and create a tab on top that we can click to go back to where we were. Depending on what we click on, we step into our selection and are exposed to more categories.

For example, if our selection is the `Point` class, we will dive inside a group of categories made for instance methods, class methods, super instance meth-

ods etc.

The interface is fully controllable through the keyboard. The user can move with `Up/Down` arrows between items or `Cmd-Shift-Up/Cmd-Shift-Down` arrows (note that on Windows and Linux `Cmd` key is the `Alt` key) through categories. At the same time, the search field has the focus, so the user can switch seamlessly between selecting items and refining the search. Pressing `Enter` on a selection opens the System Browser on that specific selected search result.

2.22 Using 'Find class' in System Browser

In the SystemBrowser you can also search for a class via its name. For example, suppose that you are looking for some unknown class that represents dates and times.

In the System Browser, click anywhere in the package pane or the class pane, and launch the Class Search window by typing `CMD-f`, or selecting `Find class` from the right-click context menu. Type `time` in the dialog box and click `OK` (or press `Enter`).

A list of classes is displayed, whose names contain the substring `time`. Choose one (say, `Time`), and the browser will show it. If you want to browse one of the others, select its name (in any text pane), and type `CMD-b`, or you can choouse `Brose` from the right-click context menu.

2.23 Finding Methods

Sometimes you can guess the name of a method, or at least part of the name of a method, more easily than the name of a class. For example, if you are interested in the current time, you might expect that there would be a method called "now", or containing "now" as a substring. But where might it be? **Spotter** and **Finder** can help you.

Spotter.

With **Spotter** you can also find methods. Either by getting a class and navigating or using category such as:

- `#implementor a method name` will display all the methods that are implemented and have the same name. For example you will get all the `do:` methods.
- `#selector a method name` will display all the selectors that matches this name

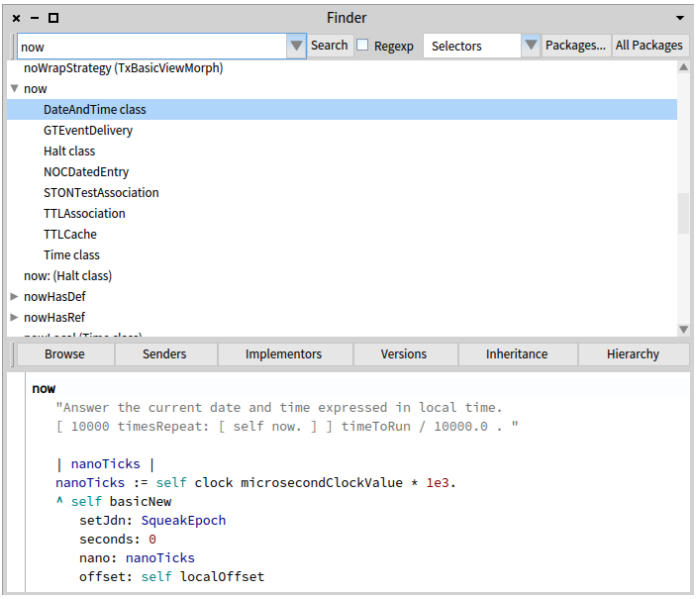

Figure 2-19 The Finder showing all classes defining a method named now.

With Finder.

Select `World Menu > Tools > Finder`. Type `now` in the top left search box, click `Search` (or just press the `Enter` key). You should see a list of results similar to the one in Figure 2-19.

The Finder will display a list of all the method names that contain the substring ”now”. To scroll to `now` itself, move the cursor to the list and type ”n”; this type-ahead trick works in all scrolling windows. Expanding the ”now” item shows you the classes that implement a method with this name. Selecting any one of them will display the source code for the implementation in the code pane on the bottom. It is also possible to search for the exact match, by typing `"now"` in the top left search bar, using quotes you will only get the exact match.

2.24 Finding Methods Using Examples

You can also open the `Finder` that is available from the `World > Tools...` menu, and type part of the name of the class and change the Selectors to Classes in the right combo box. The Finder is more useful for other types of code searches such as find methods based on examples.

At other times, you may have a good idea that a method exists, but will have no idea what it might be called. The Finder can still help! For example, sup-

pose that you would like to find a method that turns a string into upper case (for example, transforming `'eureka'` into `'EUREKA'`). We can give the inputs and expected output of a method and the Finder will try to find it for you.

The **Finder** has a really powerful functionality: you can give the receiver, arguments and expected result and the finder tries to find the corresponding message.

2.25 Trying Finder

In the Finder, select the **Examples** mode using the second combo-box (the one that shows `Selectors` by default).

Type `'eureka' . 'EUREKA'` into the search box and press the `Enter` key (don't forget the single quotes).

The **Finder** will then suggest a method that does what you were looking for, as well as display a list of classes that implement methods with the same name. In this case, it determined that the `asUppercase` method is the one that performed the operation that fit your example.

Click on the `'eureka' asUppercase --> 'EUREKA'` expression, to show the list of classes that implement that method.

An asterisk at the beginning of a line in the list of classes indicates that this method is the one that was actually used to obtain the requested result. So, the asterisk in front of `String` lets us know that the method `asUppercase` defined in the class `String` was executed and returned the result we wanted. The classes that do not have an asterisk are just other implementors of `asUppercase`, which share the method name but were *not* used to return the wanted result. So the method `Character>>asUppercase` was not executed in our example, because `'eureka'` is not a `Character` instance (but is instead a `String`).

You can also use the Finder to search for methods by arguments and results. For example, if you are looking for a method that will find the greatest common factor of two integers, you might try `25 . 35 . 5` as an example. You can also give the method finder multiple examples to narrow the search space; the help text in the bottom pane explains how.

2.26 Defining a New Method

The advent of Test Driven Development (TDD) has changed the way we write code. The idea behind TDD is that we write a test that defines the desired behaviour of our code before we write the code itself. Only then do we write the code that satisfies the test.

Suppose that our assignment is to write a method that "says something loudly and with emphasis". What exactly could that mean? What would be a good name for such a method? How can we make sure that programmers who may have to maintain our method in the future have an unambiguous description of what it should do? We can answer all of these questions by giving an example.

Our goal is to define a new method named `shout` in the class `String`. The idea is that this message should turn a string into its uppercase version as shown in the example below:

```
'No panic' shout
>>> 'NO PANIC!'
```

However, before creating the `shout` method itself, we must first create a test method! In the next section, we can use the "No Panic" example to create our test method.

2.27 Defining a New Test Method

How do we create a new method in Pharo? First, we have to decide which class the method should belong to. In this case, the `shout` method that we are testing will go in class `String`, so the corresponding test will, by convention, go in a class called `StringTest`.

First, open a browser on the class `StringTest`, and select an appropriate protocol for our method, in this case `'tests - converting'`. The highlighted text in the bottom pane is a template that reminds you what a Pharo method looks like. Delete this template code (remember, you can either click on the beginning or the end of the text, or press `CMD-a`, to "Select All"), and start typing your method. We can turn our "No Panic" code example into the test method itself:

```
testShout
  self assert: ('No panic' shout = 'NO PANIC!')
```

Once you have typed the text into the browser, notice that the right upper corner is orange. This is a reminder that the pane contains unsaved changes. So, select `Accept (s)` by right clicking in the bottom pane, or just type `CMD-s`, to compile and save your method. You should see a situation similar to the one depicted in Figure 2-20.

If this is the first time you have accepted any code in your image, you will likely be prompted to enter your name. Since many people have contributed code to the image, it is important to keep track of everyone who creates or modifies methods. Simply enter your first and last names, without any spaces.

Because there is no method called `shout` yet, the automatic code checker (Quality Assitance) will inform you that the message `shout` is sent but not

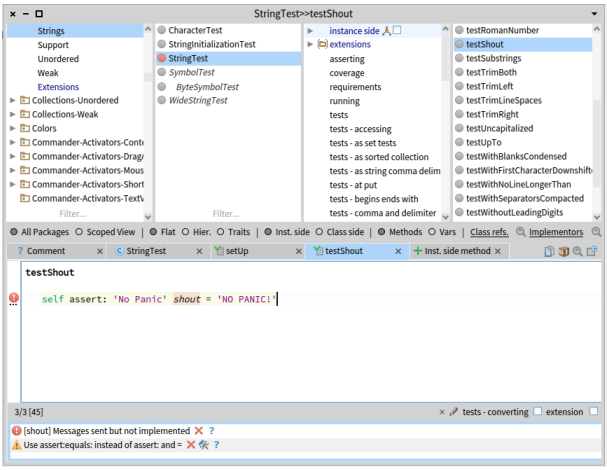

Figure 2-20 Defining a test method in the class `StringTest`.

implemented, you will see it in the lower browser pane and on the same line where you wrote the code. This can be quite useful if you have merely made a typing mistake, but in this case, we really do mean `shout`, since that is the method we are about to create. We confirm this by selecting the first option from the menu of choices.

2.28 Running Your Test Method

Run your newly created test: open the **Test Runner** from the World Menu (or press on the circle icon in front of the method name this is faster and cooler).

In the **Test Runner** the leftmost two panes are a bit like the top panes in the System Browser. The left pane contains a list of packages, but it's restricted to those packages that contain test classes.

Select `CollectionsTests-Strings` package, and the pane to the right will show all of the test classes in it, which includes the class `StringTest`. Class names are already selected, so click `Run Selected` to run all these tests.

You should see the upper right pane turn red, which indicates that there was an error in running the tests. The list of tests that gave rise to errors is shown in the bottom right pane. As you can see, the method `StringTest>>test-Shout` is the culprit. (Note that `StringTest>>testShout` is the Pharo way of identifying the `testShout` method of the `StringTest` class.) If you click on that method in the bottom right pane, the erroneous test will run again, this time in such a way that you see the error happen: `Instenace of ByteString did not understand #shout` (see Figure 2-21).

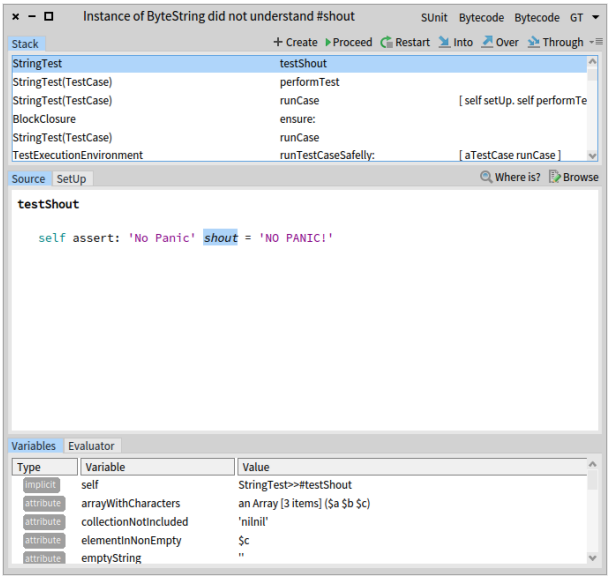

Figure 2-21 Looking at the error in the debugger.

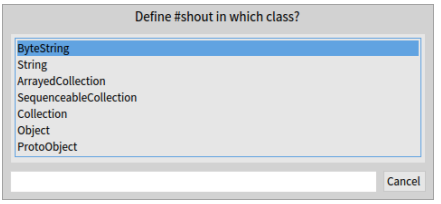

Figure 2-22 Pressing the Create button in the debugger prompts you to select in which class to create the new method.

The window that opens with the error message is the Pharo debugger. We will look at the debugger and how to use it in Chapter: The Pharo Environment.

2.29 Implementing the Tested Method

The error is, of course, exactly what we expected: running the test generates an error because we have not yet written a method that tells strings how to shout. Nevertheless, it's good practice to make sure that the test fails because this confirms that we have set up the testing machinery correctly and that the new test is actually being run. Once you have seen the error, you can Abandon the running test, which will close the debugger window.

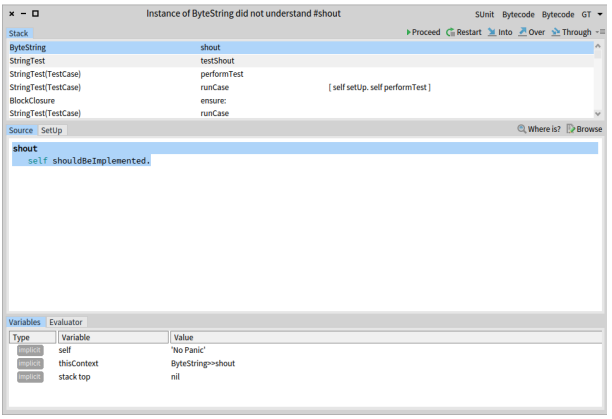

Figure 2-23 The automatically created `shout` method waiting for a real definition.

Coding in the Debugger

Instead of pressing `Abandon`, you can define the missing method using the `Create` button right in the debugger. This will prompt you to select a class in which to define the new method (see Figure 2-22), then prompt you to select a protocol for that method, and finally take you to a code editor window in the debugger, in which you can edit the code for this new method. Note that since the system cannot implement the method for you, it creates a generic method that is tagged as to be implemented (see Figure 2-23).

Now let's define the method that will make the test succeed! Right inside the debugger edit the `shout` method with this definition:

```
shout
  ^ self asUppercase,'!'
```

The comma is the string concatenation operation, so the body of this method appends an exclamation mark to an upper-case version of whatever String object the shout message was sent to. The ^ tells Pharo that the expression that follows is the answer to be returned from the method, in this case the new concatenated string.

When you've finished implementing the method, do not forget to compile it using `CMD-s` and you can press `Proceed` and continue with the tests. Note that `Proceed` simply continues on running the test suite, and does not re-run the failed method.

Does this method work?

Let's run the tests and see. Click on `Run Selected` again in the Test Runner, and this time you should see a green bar and text indicating that all of the

tests ran with no failures and no errors. When you get to a green bar, it's a good idea to save your work by saving the image (`World Menu > Pharo > Save`), and take a break. So, do that right now!

2.30 Chapter Summary

This chapter has introduced you to the Pharo environment and shown you how to use some of the major tools, such as the System Browser, Spotter, the Finder, the Debugger, and the Test Runner. You have also seen a little of Pharo's syntax, even though you may not understand it all yet.

- A running Pharo system consists of a *virtual machine*, a `.sources` file, and `.image` and `.changes` files. Only these last two change, as they record a snapshot of the running system.
- When you open a Pharo image, you will find yourself in exactly the same state (i.e., with exactly the same running objects) that you had when you last saved that image.
- You can click on the Pharo background to bring up the **World Menu** and launch various tools.
- A **Playground** is a tool for writing and evaluating snippets of code. You can also use it to store arbitrary text.
- You can use keyboard shortcuts on text in the playground, or any other tool, to evaluate code. The most important of these are `Do it` (`CMD-d`), `Print it` (`CMD-p`), `Inspect it` (`CMD-i`), and `Browse it` (`CMD-b`).
- The **System Browser** is the main tool for browsing Pharo code and for developing new code.
- The **Test runner** is a tool for running unit tests, and aids in Test Driven Development.
- The **Debugger** allows you to examine errors and exceptions (such as errors or failures encountered when running tests). You can even create new methods right in the debugger.

CHAPTER 3

Developing a simple counter

To get started in Pharo, we invite you to implement a simple counter by following the steps given below. In this exercise you will learn how to create packages classes, method, instances. You will learn how to define tests and more. This simple tutorial covers most of the important actions that we do when developing in Pharo. You can also watch the companion videos available in the Pharo mooc at http://mooc.pharo.org: they illustrate this tutorial in a more lively manner.

Note that the development flow promoted by this little tutorial is *traditional* in the sense that you will define a package, a class, *then* define its instance variable *then* define its methods *and* finally execute it. We show also how you can save your code on git hosting services such as github using Iceberg. Now in Pharo, developers often follows a *totally* different style (that we call live coding or Xtreme TDD) where they execute an expression that raises errors and they code in the debugger and let the system define some instance variables and methods on the fly for them.

Once you will have finish this tutorial, you will feel more confident with Pharo and we strongly suggest you to try the other style.

3.1 Our use case

Here is our use case: We want to be able to create a counter, increment it twice, decrement it and check that its value is correct. It looks like this little use case will fit perfectly a unit test - you will define one later.

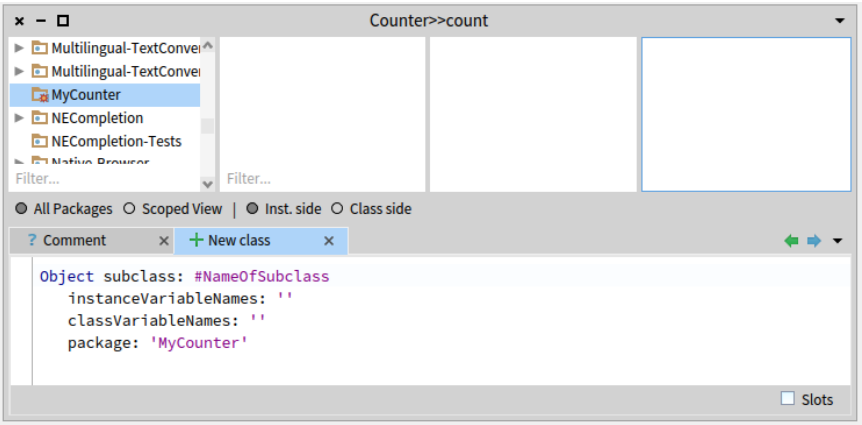

Figure 3-1 Package created and class creation template.

```
| counter |
counter := Counter new.
counter increment; increment.
counter decrement.
counter count = 1
```

Now we will develop all the mandatory class and methods to support this scenario.

3.2 Create your own class

In this part, you will create your first class. In Pharo, a class is defined in a package. You will create a package then a class. The steps we will do are the same ones every time you create a class, so memorize them well.

3.3 Create a package and class

Using the Browser create a package. The system will ask you a name, write `MyCounter`. This new package is then created and added to the list. Figure 3-1 shows the result of creating such a package.

Create a class.

Creating a class requires five steps. They consist basically in editing the class definition template to specify the class you want to create.

- Superclass Specification. First, you should replace the word `NameOf-Superclass` with the word `Object`. Thus, you specify the superclass of

the class you are creating. Note that this is not always the case that `Object` is the superclass, since you may to inherit behavior from a class specializing already `Object`.

- Class Name. Next, you should fill in the name of your class by replacing the word `NameOfClass` with the word `Counter`. Take care that the name of the class starts with a capital letter and that you do not remove the #sign in front of `NameOfClass`. This is because the class we want to create does not exist yet, so we have to give its name, and we use a Symbol (a unique string in Pharo) to do so.
- Instance Variable Specification. Then, you should fill in the names of the instance variables of this class. We need one instance variable called `count`. Take care that you leave the string quotes!
- Class Variable Specification. As we do not need any class variable make sure that the argument for the class instance variables is an empty string `classInstanceVariableNames: ''`.

You should get the following class definition.

```
Object subclass: #Counter
      instanceVariableNames: 'count'
      classVariableNames: ''
      package: 'MyCounter'
```

Now we should compile it. We now have a filled-in class definition for the class `Counter`. To define it, we still have to *compile* it. Therefore, select the accept menu item. The class `Counter` is now compiled and immediately added to the system.

Figure 3-2 illustrates the resulting situation that the browser should show.

The tool runs automatically some code critic and some of them are just inaccurate, so do not care for now.

As we are disciplined developers, we add a comment to `Counter` class by clicking Comment button. You can write the following comment:

```
Counter is a simple concrete class which supports incrementing and
    decrementing a counter.
Its API is
- decrement, increment
- count
Its creation API is message startingAt:
```

Select menu item 'accept' to store this class comment in the class.

Figure 3-3 shows the class with its comment.

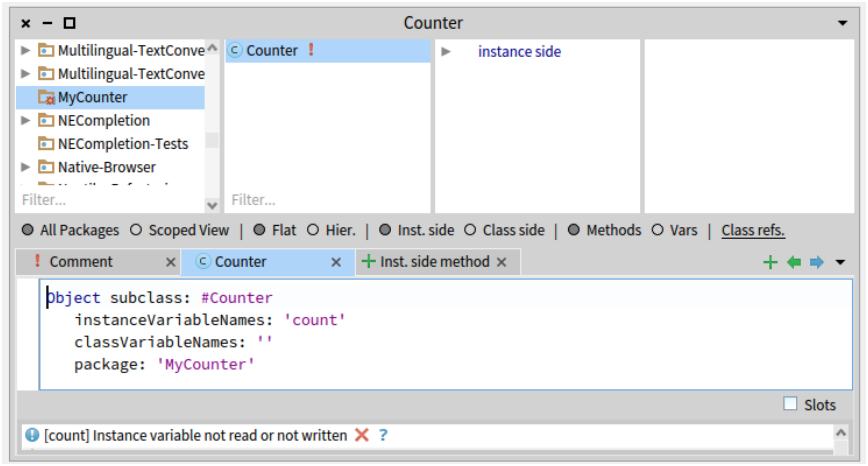

Figure 3-2 Class created: It inherits from `Object` class and has one instance variable named `count`.

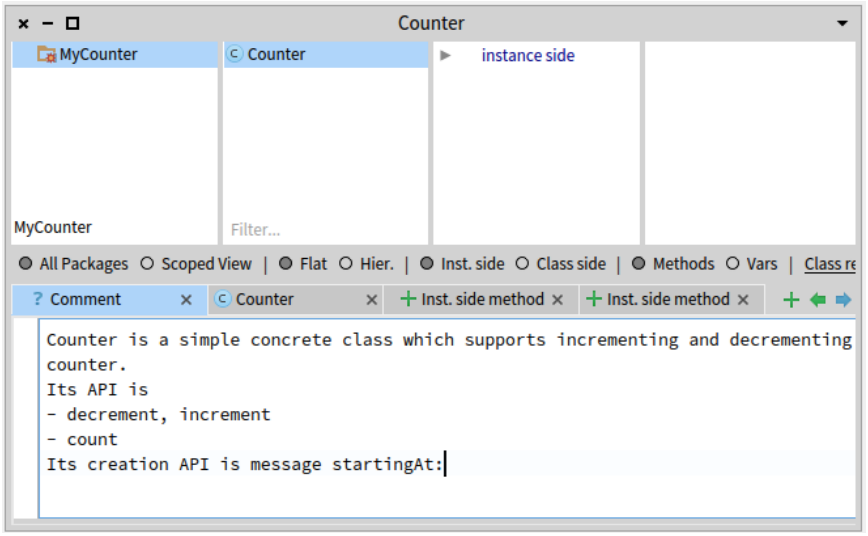

Figure 3-3 Counter class has now a comment! Well done.

3.4 Define protocols and methods

In this part you will use the browser to learn how to add protocols and methods.

The class we have defined has one instance variable named `count`. You should remember that in Pharo, (1) everything is an object, (2) that instance variables are private to the object, and (3) that the only way to interact with an object is by sending messages to it.

Therefore, there is no other mechanism to access the instance variable values from outside an object than sending a message to the object. What you can do is to define messages that return the value of the instance variable. Such methods are called *accessors*, and it is a common practice to always define and use them. We start to create an accessor method for our instance variable `count`.

A method is usually sorted into a protocol. These protocols are just a group of methods without any language semantics, but convey important navigation information for the reader of your class. Although protocols can have any name, Pharo programmers follow certain conventions for naming these protocols. If you define a method and are not sure what protocol it should be in, first go through existing code and try to find a fitting name.

3.5 Create a method

Now let us create the accessor methods for the instance variable `count`. Start by selecting the class `Counter` in a browser, and make sure the you are editing the instance side of the class (i.e., we define methods that will be sent to instances) by deselecting the Class side radio button.

Click on the instance method tab and define your method.

Figure 3-4 shows the method editor ready to define a method.

As a general hint, double click at the end of or beginning of the text and start typing your method: this automatically replace your Replace the template with the following method definition:

```
count
   ^ count
```

This defines a method called `count`, taking no arguments, having a method comment and returning the instance variable `count`. Then choose *accept* in the menu to compile the method. The method is automatically categorized in the protocol *accessing*.

Figure 3-5 shows the state of the system once the method is defined.

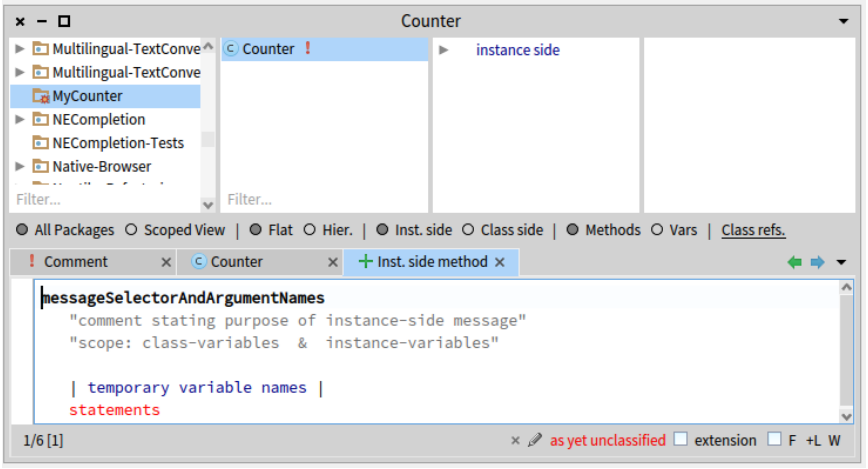

Figure 3-4 The method editor selected and ready to define a method.

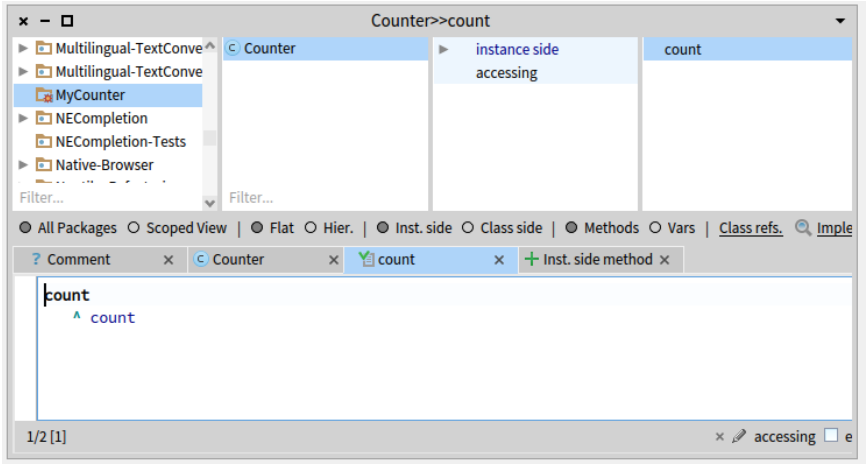

Figure 3-5 The method `count` defined in the protocol *accessing*.

You can now test your new method by typing and evaluating the next expression in a Playground, or any text editor.

```
Counter new count
>>> nil
```

This expression first creates a new instance of `Counter`, and then sends the message `count` to it. It retrieves the current value of the counter. This should return `nil` (the default value for non-initialised instance variables). Afterwards we will create instances with a reasonable default initialisation value.

3.6 Adding a setter method

Another method that is normally used besides the accessor method is a so-called setter method. Such a method is used to change the value of an instance variable from a client. For example, the expression `Counter new count: 7` first creates a new `Counter` instance and then sets its value to 7:

The snippets shows that the counter effectively contains its value.

```
| c |
c := Counter new count: 7.
c count
>>> 7
```

This setter method does not currently exist, so as an exercise write the method `count:` such that, when invoked on an instance of `Counter`, instance variable is set to the argument given to the message. Test your method by typing and evaluating the expression above.

3.7 Define a Test Class

Writing tests is an important activity that will support the evolution of your application. Remember that a test is written *once and executed million* times. For example if we have turned the expression above into a test we could have checked automatically that our new method is correctly working.

To define a test case we will define a class that inherits from `TestCase`. Therefore define a class named `CounterTest` as follows:

```
TestCase subclass: #CounterTest
      instanceVariableNames: ''
      classVariableNames: ''
      package: 'MyCounter'
```

Now we can write a first test by defining one method. Test methods should start with *text* to be automatically executed by the TestRunner or when you press on the icon of the method. Now to make sure that you understand in which class we define the method we prefix the method body with the class

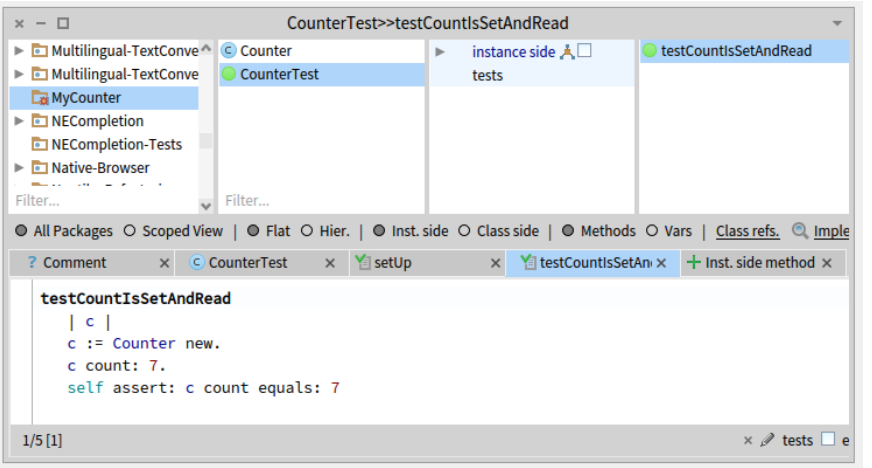

Figure 3-6 A first test is defined and it passes.

name and >>. `CounterTest>>` means that the method is defined in the class `CounterTest`.

Figure 3-6 shows the definition of the method `testCountIsSetAndRead` in the class `CounterTest`.

Define the following method. It first creates an instance, sets its value and verifies that the value is correct. The message `assert:equals:` is a special message verifying if the test passed or not.

```
CounterTest >> testCountIsSetAndRead
   | c |
   c := Counter new.
   c count: 7.
   self assert: c count equals: 7
```

Verify that the test passes by executing either pressing the icon in front of the method (as shown by Figure 3-6) or using the TestRunner available in the Tools menus (selecting your package). Since you have a first green test. This is a good moment to save your work.

3.8 Saving your code on git with Iceberg

With Iceberg, we will show you how to save your code locally then later we will push it to GitHub.

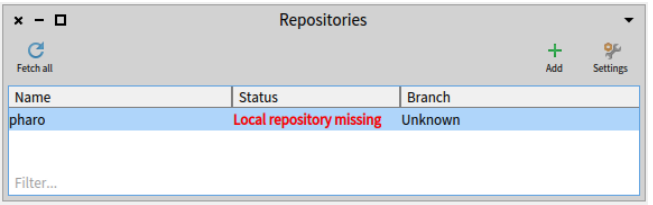

Figure 3-7 Iceberg *Repositories* browser on a fresh image indicates that if you want to version modifications to Pharo itself you will have to tell Iceberg where the Pharo clone is located. But you do not care.

Open Iceberg.

You should the situation depicted by Figure 3-7 which shows the top level Iceberg pane. It shows that for now you do not have defined nor loaded any project. It shows the Pharo project and indicates that it could not find its local repository by displaying 'Local repository missing'. You do not have to worry about the Pharo project or repository if you do not want to contribute to Pharo. So just go ahead. Since you do not plan to modify and version the Pharo system code, you do not have to worry.

Add and configure a project.

Press the iconic button Add to create a new project. Pick up 'New Repository' and you should get a configuration pane similar to the one of Figure 3-8. Here we define the Project named 'MyCounter', give a directory on our disk and we indicate that the source should be in the subfolder `src`.

Add your package to the project.

Once added, Iceberg *Working copy* browser should show you an empty pane because you did not add any package to your project. Click on the Add package iconic button and select the package MyCounter as shown by Figure 3-9.

Commit your changes.

Once you package is added, Iceberg shows you that you did not commit your code as shown in Figure 3-11. Press the Commit iconic button. Iceberg will show you all the changes that are about to be saved (Figure 3-11). Enter a commit message and commit

Code saved.

Once you have commited, Iceberg indicates that your system and local repository are in sync.

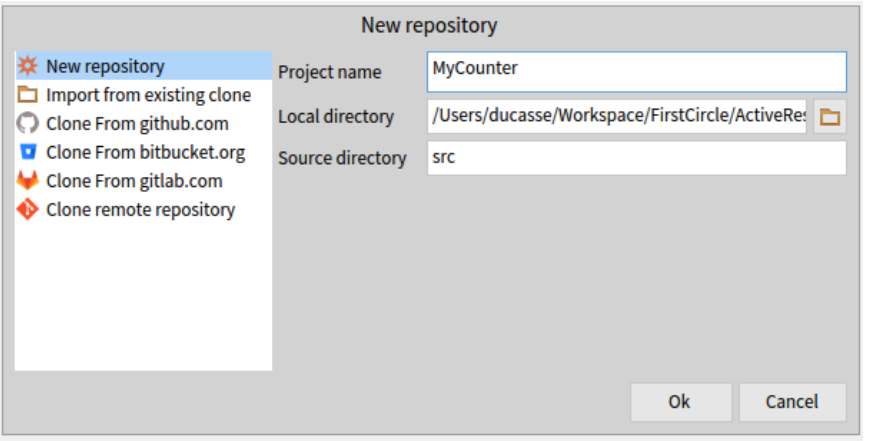

Figure 3-8 Add and create a project named MyCounter and with the `src` sub-folder.

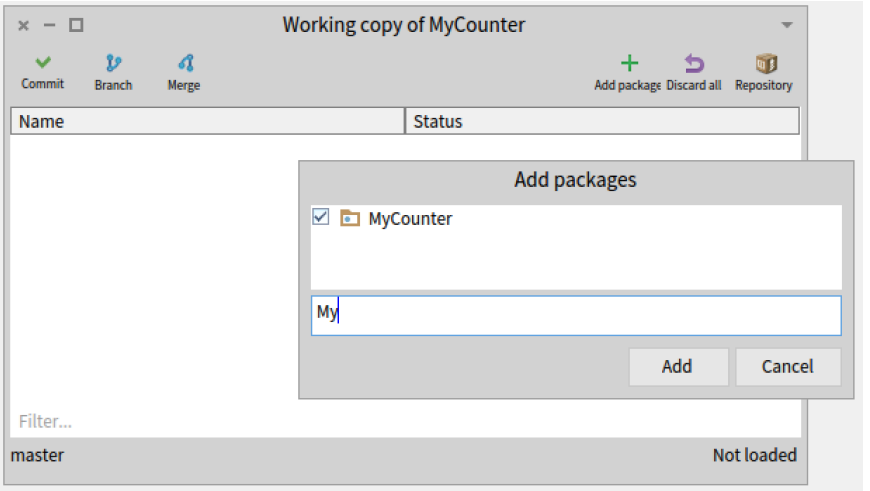

Figure 3-9 Selecting the Add package iconic button, add your package MyCounter to your project.

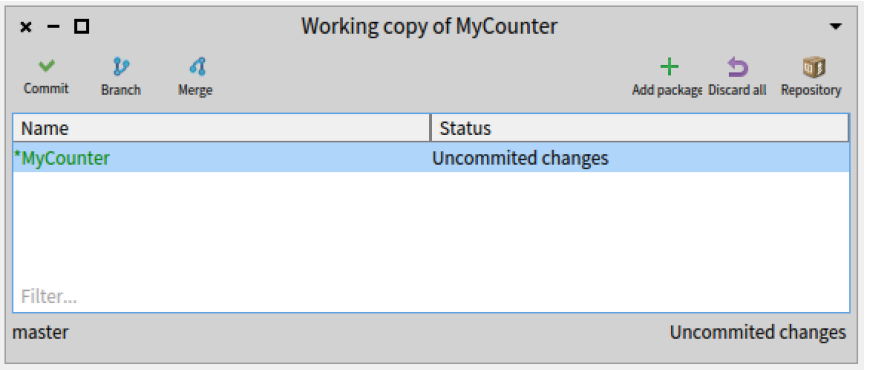

Figure 3-10 Now Iceberg shows you that you did not commit your code.

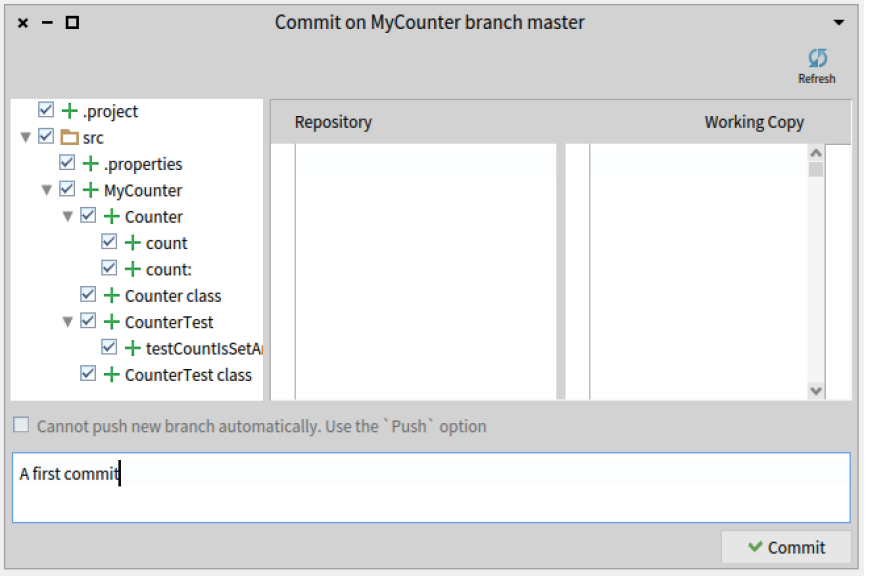

Figure 3-11 Iceberg shows you the changes about to be commited.

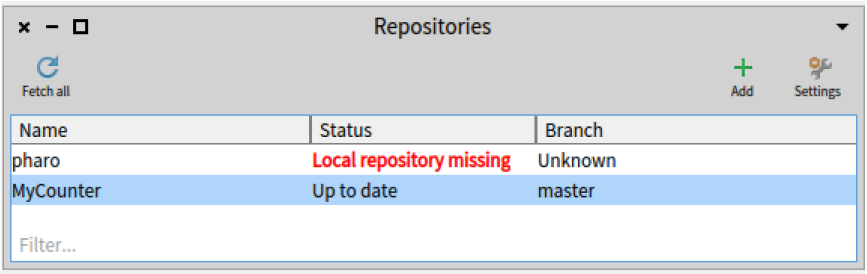

Figure 3-12 Once you save your change, Iceberg shows you that .

3.9 Adding more messages

Before implementing the following messages we define first a test. We define one test for the method `increment` as follows:

```
CounterTest >> testIncrement
   | c |
   c := Counter new.
   c count: 0 ; increment; increment.
   self assert: c count equals: 2
```

- Propose a definition for the method `increment`.
- Define a test and method for the method `decrement`.
- Implement the following methods `increment` and `decrement` in the protocol 'operation'.
- Implement also a new test method for the method `decrement`.

```
Counter >> increment
   count := count + 1
```

```
Counter >> decrement
   count := count - 1
```

Run your tests they should pass (as shown in Figure 3-13). Again this is a good moment to save your work. Saving at point where tests are green is always a good process. To save your changes, you just have to commit them.

3.10 Instance initialization method

Right now the initial value of our counter is not set as the following expression shows it.

```
Counter new count
>>> nil
```

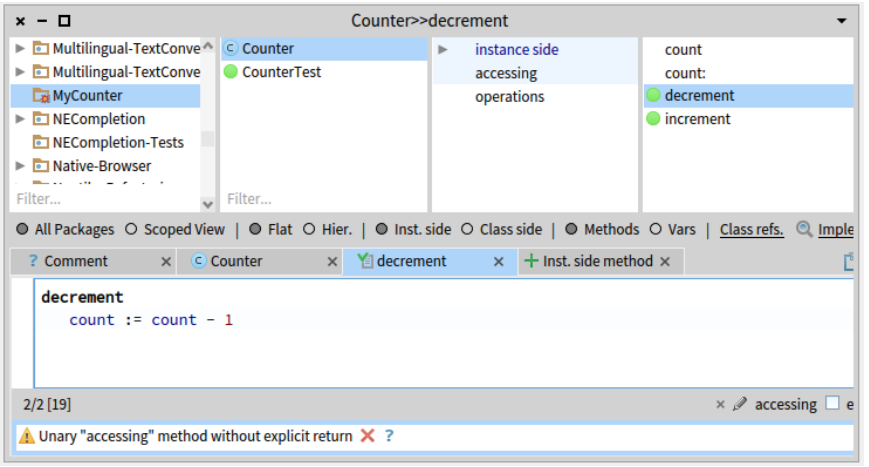

Figure 3-13 Class with more green tests.

Let us write a test checking that a newly created instance has 0 as a default value.

```
CounterTest >> testInitialize
   self assert: Counter new count equals: 0
```

If you run it, it will turn yellow indicating a failure (a situation that you anticipated but that is not correct) - by opposition to an error which is an anticipated situation leading to failed assertion.

3.11 Define an initialize method

Now we have to write an initialization method that sets a default value of the `count` instance variable. However, as we mentioned the `initialize` message is sent to the newly created instance. This means that the `initialize` method should be defined at the instance side as any method that is sent to an instance of `Counter` (like `increment`) and `decrement`. The `initialize` method is responsible to set up the instance variable default values.

Therefore at the instance side, you should create a protocol `initialization`, and create the following method (the body of this method is left blank. Fill it in!).

```
Counter >> initialize
  "set the initial value of the value to 0"
  ...
  Fill me please!!!
```

Now create a new instance of class `Counter`. Is it initialized by default? The following code should now work without problem:

```
Counter new increment count
>>> 1
```

and the following one should return 2

```
Counter new increment; increment; count
>>> 2
```

But better write a test since we will execute it all the time.

```
TestCounter >> testCounterWellInitialized
  self
    assert: (Counter new increment; increment; count)
    equals: 2
```

Again save your work before starting the next step.

3.12 Define a new instance creation method

We would like to show you the difference between an instance method (i.e. sent to instances) and a class method (i.e., to a class). In fact the only difference is the place to define them. An instance method is defined in the instance side of Code Browser while class methods are defined on the class side (Pressing the button Class).

Define a different instance creation method named `startingAt:`. This method receives an integer as argument and returns an instance of `Counter` with the specified value.

Let us define a test:

```
TestCounter >> testCounterStartingAt5
  self assert: (Counter startingAt: 5) count equals: 5
```

Here the message `startingAt:` is sent to the class `Counter` itself.

Your implementation should look like

```
Counter class >> startingAt: anInteger
  ^ self new count: anInteger.
```

Note that `self` in such method refers to the class `Counter` itself.

Let us write another test to check that everything is working.

```
CounterTest >> testAlternateCreationMethod
  self assert: ((Counter startingAt: 19) increment ; count) equals:
    20
```

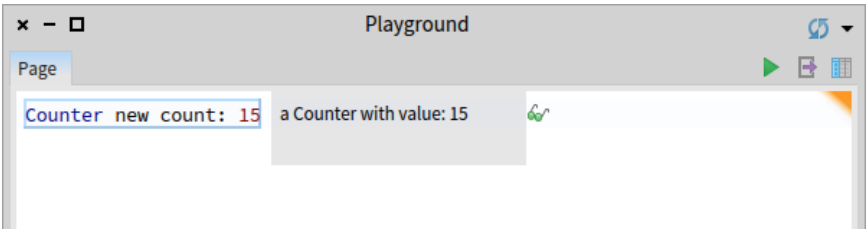

Figure 3-14 Better description.

3.13 Better object description

When you open an inspect (putting a `self halt` inside a method definition) you obtain an inspector or when you select the expression `Counter new` and print its result (using the Print it menu of the editor) you obtain a simple string `'a Counter'`.

```
Counter new
>>> a Counter
```

We would like to get a much richer information for example knowing the counter value. Implement the following methods in the protocol `printing`

```
Counter >> printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: ' with value: ', count printString.
```

Note that the method `printOn:` is used when you print an object using print it (See Figure 3-14) or click on `self` in an inspector.

We let you define a method for this method. A tip send the message `printString` to `Counter new` to get its string representation.

```
  Counter new printString
>>> a Counter with value: 0
```

3.14 Saving your code on a remote server

Up until now you saved your code on your local disc. We will now show how you can save your code on a remote repository such as the one you can create on GitHub http://github.com or Gitlab.

Create a project on the remote server.

First you should create a project with the same name than the one of your project.

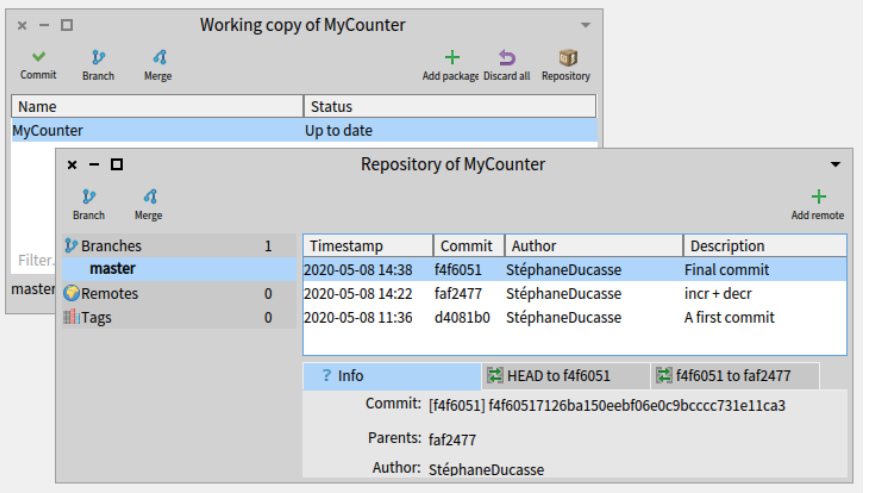

Figure 3-15 A *Repository* browser opened on your project.

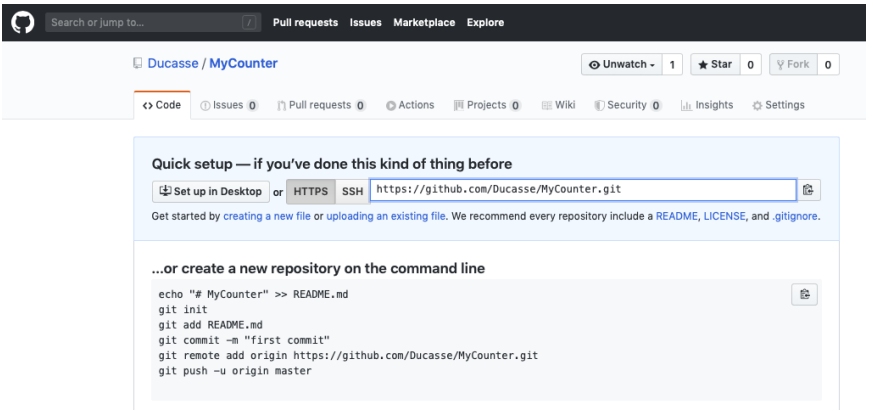

Figure 3-16 GitHub HTTPS address our our project.

Add a remote repository in HTTPS access.

Clicking on the Repository iconic button of the *Working copy* browser, you get access to the *Repository* browser open on your project as show in Figure 3-15.

Then you just have to add a remote repository using the Add remote iconic button of the *Repository* browser. For this we will use the project identification address given by the remote browser. Since we decided to use HTTPS we use https://github.com/Ducasse/MyCounter.git as address as shown in Figure 3-16 and Figure 3-17.

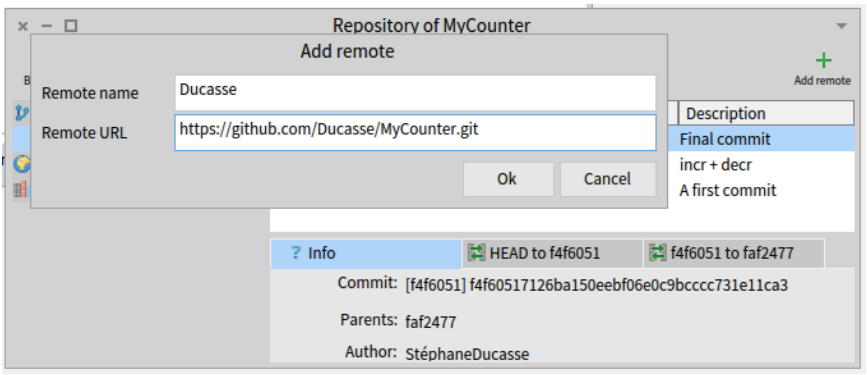

Figure 3-17 Using the GitHub HTTPS address.

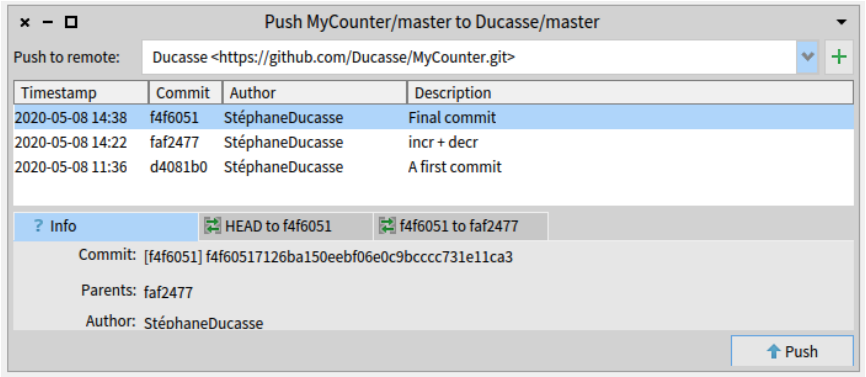

Figure 3-18 Commits sent to the remote repository.

Push.

As soon as you add a valid server address, Iceberg will show a little red indication on the Push iconic button. This shows that you have changes in your local repository that have not being pushed to your remote repository. Now you just have to press the Push iconic button. Iceberg will show you the commits that will be pushed to the server as shown in Figure 3-18.

Now you fully saved your code and you will be able to reload from another machine or location. This will enable you to work remotely and collaborately.

3.15 Conclusion

In this tutorial you learned how to define packages, classes, methods, and define tests. The flow of programming that we chose for this first tutorial is similar to most of programming languages. In Pharo you can use a different flow that is based on defining a test first, executing it and when the execution raises error to define the corresponding classes, methods, and instance variable often from inside the debugger. We suggest you now to redo the exercise following the second companion video.

CHAPTER 4

A first application

In this chapter, we will develop a simple game: LightsOut (http://en.wikipedia.org/wiki/Lights_Out_(game)). Along the way we will show most of the tools that Pharo programmers use to construct and debug their programs, and show how programs are shared with other developers. We will see the browser, the object inspector, the debugger and the way to version code.

In Pharo you can develop in a traditional way, by defining a class, then its instance variables, then its methods. However, in Pharo your development flow can be much more productive than that! You can define instance variables and methods on the fly. You can also code in the debugger using the exact context of currently executed objects. This chapter will sketch such alternate way and show you how you can be really productive.

We will code the game but doing so we will make mistakes and we will show you how we recover from these mistakes. So this may be a bit frustrating for you and more boring for us to describe but this is a key aspect of programming. We have to show how to handle errors and find bugs.

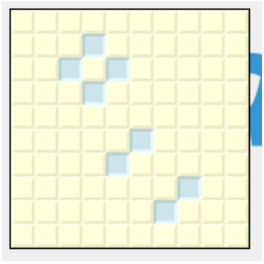

Figure 4-1 The Lights Out game board.

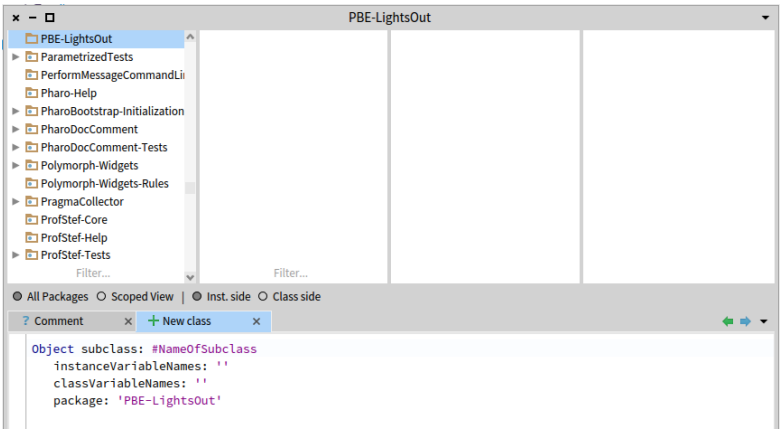

Figure 4-2 Create a Package and class template.

4.1 The Lights Out game

To show you how to use Pharo's programming tools, we will build a simple game called **Lights Out**. The game board consists of a rectangular array of light yellow cells. When you click on one of the cells, the four surrounding cells turn blue. Click again, and they toggle back to light yellow. The object of the game is to turn blue as many cells as possible.

Lights Out is made up of two kinds of objects: the game board itself, and 100 individual cell objects. The Pharo code to implement the game will contain two classes: one for the game and one for the cells. We will now show you how to define these classes using the Pharo programming tools.

4.2 Creating a new Package

We have already seen the browser in Chapter : A Quick Tour of Pharo where we learned how to navigate to packages, classes and methods, and saw how to define new methods. Now we will see how to create packages and classes.

From the `World` menu, open a **System Browser**. Right-click on an existing package in the Package pane and select `New package` from the menu. Type the name of the new package (we use `PBE-LightsOut`) in the dialog box and click `OK` (or just press the return key). The new package is created, and positioned alphabetically in the list of packages (see Figure 4-2).

Hints: You can type `PBE` in the filter to get your package filtered out the other ones (See Figure 4-3).

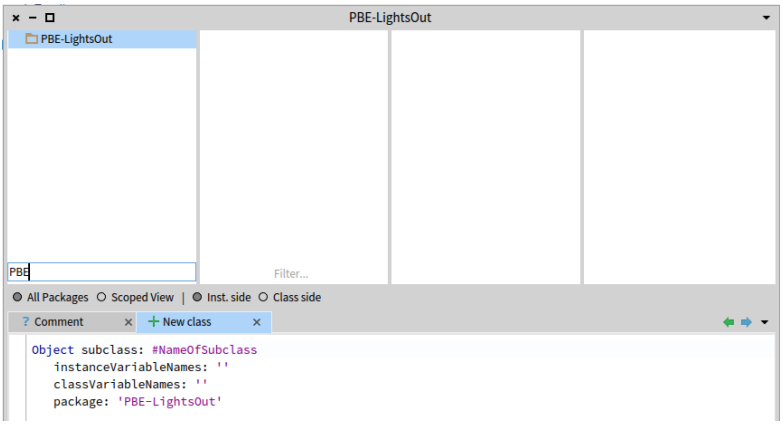

Figure 4-3 Filtering our package to work more efficiently.

Listing 4-4 `LOCell` class definition

```
SimpleSwitchMorph subclass: #LOCell
  instanceVariableNames: 'mouseAction'
  classVariableNames: ''
  package: 'PBE-LightsOut'
```

4.3 Defining the class `LOCell`

At this point there are, of course, no classes in the new package. However, the main editing pane displays a template to make it easy to create a new class (see Figure 4-3).

This template shows us a Pharo expression that sends a message to a class called `Object`, asking it to create a subclass called `NameOfSubClass`. The new class has no variables, and should belong to the category (package) `PBE-LightsOut`.

4.4 Creating a new class

We simply edit the template to create the class that we really want. Modify the class creation template as follows:

- Replace `Object` with `SimpleSwitchMorph`.
- Replace `NameOfSubClass` with `LOCell`.
- Add `mouseAction` to the list of instance variables.

You should get the following class definition:

This new definition consists of a Pharo expression that sends a message to the existing class `SimpleSwitchMorph`, asking it to create a subclass called

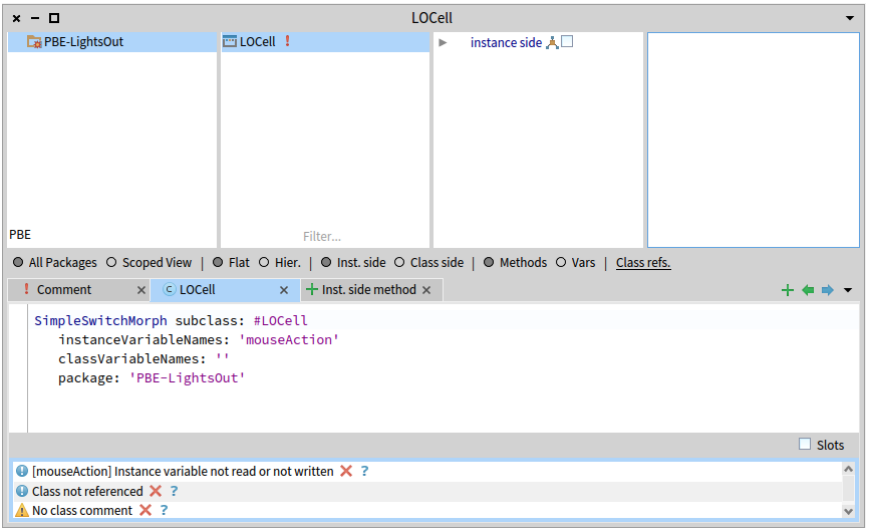

Figure 4-5 The newly-created class LOCell.

`LOCell`. (Actually, since `LOCell` does not exist yet, we passed the symbol `#LOCell` as an argument, representing the name of the class to create.) We also tell it that instances of the new class should have a `mouseAction` instance variable, which we will use to define what action the cell should take if the mouse should click on it.

At this point you still have not created anything. Note that the top right of the panel changed to orange. This means that there are unsaved changes. To actually send this subclass message, you must save (accept) the source code. Either right-click and select `Accept`, or use the shortcut `CMD-s` (for "Save"). The message will be sent to `SimpleSwitchMorph`, which will cause the new class to be compiled. You should get the situation depicted in Figure 4-5.

Once the class definition is accepted, the class is created and appears in the class pane of the browser (see Figure 4-5). The editing pane now shows the class definition. Below you get the Quality Assistant's feedback: It runs automatically quality rules on your code and reports them.

4.5 About comments

Pharoers put a very high value on the readability of their code, but also good quality comments.

Listing 4-6 Initializing instance of LOCell

```
initialize
  super initialize.
  self label: ''.
  self borderWidth: 2.
  bounds := 0 @ 0 corner: 16 @ 16.
  offColor := Color paleYellow.
  onColor := Color paleBlue darker.
  self useSquareCorners.
  self turnOff
```

Method comments.

People have the tendency to believe that it is not necessary to comment well written methods: it is plain wrong and encourages sloppiness. Of course, bad code should renamed and refactored. Obviously commenting trivial methods makes no sense. A comment should not be the code written in english but an explanation of what the method is doing, its context, or the rationale behind its implementation. When reading a comment, the reader should be comforted that his hypotheses are correct.

Class comments.

For the class comment, the Pharo class comment template gives a good idea of a strong class comment. Read it! It is based on CRC for Class Responsibility Collaborators. So in a nutshell the comments state the responsibility of the class in a couple of sentences and how it collaborates with other classes to achieve this responsibilities. In addition we can state the API (main messages an object understands), give an example (usually in Pharo we define examples as class methods), and some details about internal representation or implementation rationale.

Select the comment button and define a class comment following this template

4.6 Adding methods to a class

Now let's add some methods to our class. Select the `Inst. side method` tab next to the class definition tab. You will see a template for method creation in the editing pane. Select the template text, and replace it by the following (do not forget to compile it):

Note that the characters `''` on line 3 are two separate single quotes with nothing between them, not a double quote! `''` denotes the empty string. Another way to create an empty string is `String new.` Do not forget to compile this method Pharo using the **accept** menu item (CMD-s/Option-s).

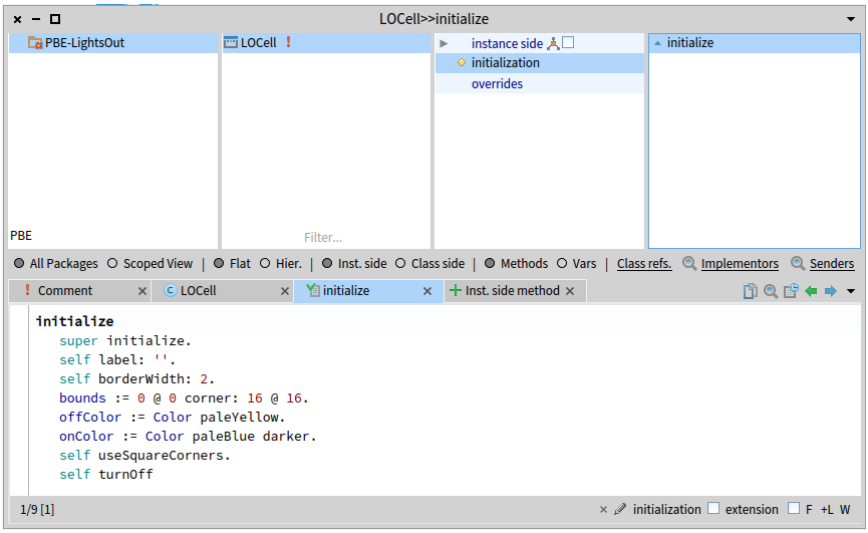

Figure 4-7 The newly-created method `initialize`.

Initialize methods.

Notice that the method is called `initialize`. The name is very significant! By convention, if a class defines a method named `initialize`, it is called right after the object is created. So, when we execute `LOCell new`, the message `initialize` is sent automatically to this newly created object. `initialize` methods are used to set up the state of objects, typically to set their instance variables; this is exactly what we are doing here.

Invoking superclass initialization.

The first thing that this method does (line 2) is to execute the `initialize` method of its superclass, `SimpleSwitchMorph`. The idea here is that any inherited state will be properly initialized by the `initialize` method of the superclass. It is always a good idea to initialize inherited state by sending `super initialize` before doing anything else. We don't know exactly what `SimpleSwitchMorph`'s `initialize` method will do (and we don't care), but it's a fair bet that it will set up some instance variables to hold reasonable default values. So we had better call it, or we risk starting in an unclean state.

The rest of the method sets up the state of this object. Sending `self label: ''`, for example, sets the label of this object to the empty string.

About point and rectangle creation.

The expression `0@0 corner: 16@16` probably needs some explanation. `0@0` represents a `Point` object with `x` and `y` coordinates both set to `0`. In fact, `0@0` sends the message `@` to the number `0` with argument `0`. The effect will be that the number `0` will ask the `Point` class to create a new instance with coordinates `(0,0)`. Now we send this newly created point the message `corner: 16@16`, which causes it to create a `Rectangle` with corners `0@0` and `16@16`. This newly created rectangle will be assigned to the `bounds` variable, inherited from the superclass.

Note that the origin of the Pharo screen is the top left, and the `y` coordinate increases downwards.

About the rest.

The rest of the method should be self-explanatory. Part of the art of writing good Pharo code is to pick good method names so that the code can be read like a kind of pidgin English. You should be able to imagine the object talking to itself and saying "Self, use square corners!", "Self, turn off!".

Notice that there is a little green arrow next to your method (see Figure 4-7). This means the method exists in the superclass and is overridden in your class.

4.7 Inspecting an object

You can immediately test the effect of the code you have written by creating a new `LOCell` object and inspecting it: Open a `Playground`, type the expression `LOCell new`, and **Inspect** it (using the menu item with the same name).

The left-hand column of the inspector shows a list of instance variables and the value of the instance variable is shown in the right column (see Figure 4-8).

If you click on an instance variable the inspector will open a new pane with the detail of the instance variable (see Figure 4-9).

Executing expressions.

The bottom pane of the inspector is a mini-playground. It's useful because in this playground the pseudo-variable `self` is bound to the object selected.

Go to that Playground at the bottom of the pane and type the text `self bounds: (200@200 corner: 250@250)` **Do it**. To refresh the values, click on the `update` button (the blue little circle) at the top right of the pane. The `bounds` variable should change in the inspector. Now type the text `self openInWorld` in the mini-playground and **Do it**.

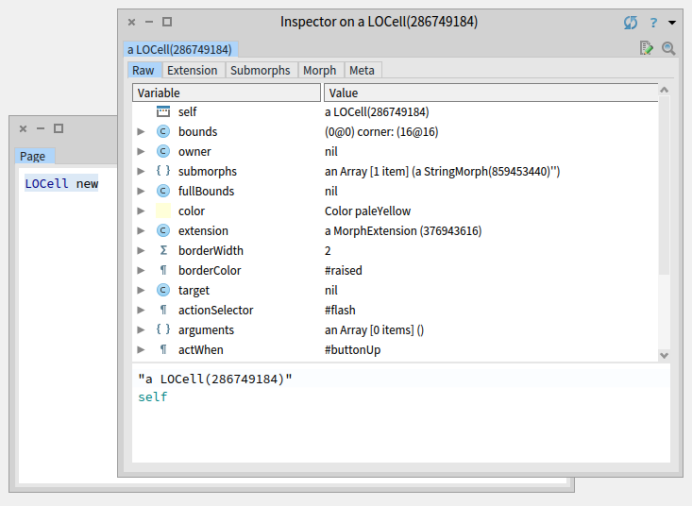

Figure 4-8 The inspector used to examine a `LOCell` object.

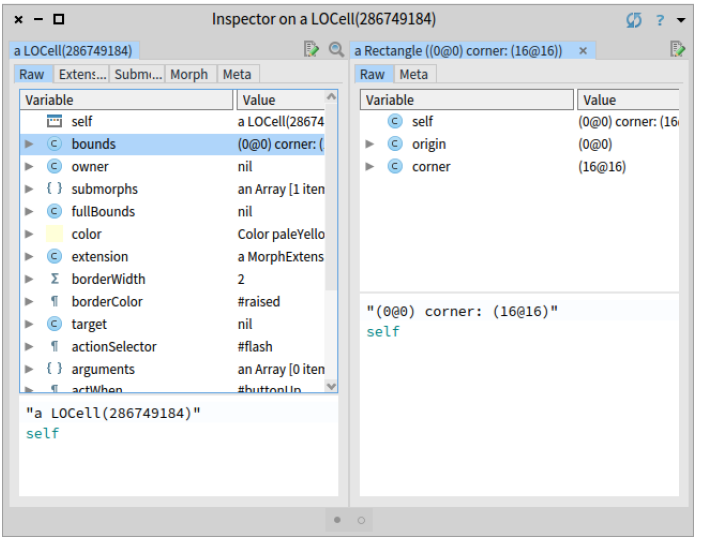

Figure 4-9 When we click on an instance variable, we inspect its value (another object).

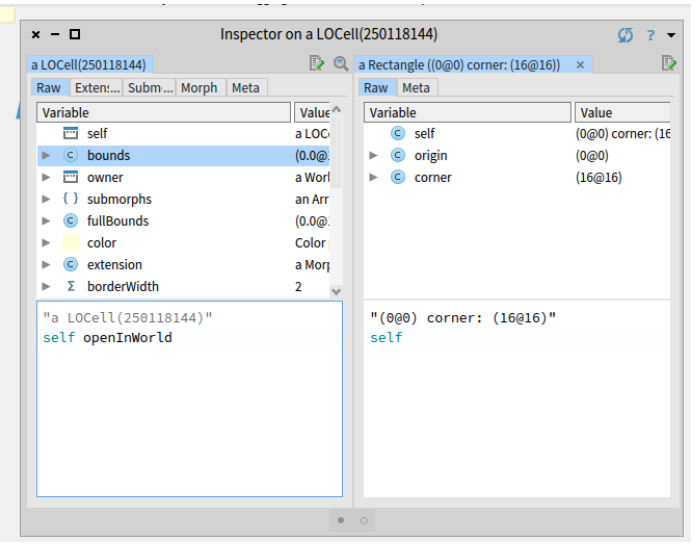

Figure 4-10 An LOCell open in world.

Listing 4-11 Defining the LOGame class

```
BorderedMorph subclass: #LOGame
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-LightsOut'
```

The cell should appear near the top left-hand corner of the screen (as shown in Figure 4-10) and exactly where its bounds say that it should appear. Meta-click on the cell to bring up the Morphic halo. Move the cell with the brown (next to top-right) handle and resize it with the yellow (bottom-right) handle. Notice how the bounds reported by the inspector also change. (You may have to click refresh to see the new bounds value.) Delete the cell by clicking on the x in the pink handle.

4.8 Defining the class LOGame

Now let's create the other class that we need for the game, which we will name `LOGame`.

Make the class definition template visible in the browser main window. Do this by clicking on the package name (or right-clicking on the Class pane and selecting `Add Class`). Edit the code so that it reads as follows, and **Accept** it.

Here we subclass `BorderedMorph`. `Morph` is the superclass of all of the graphical shapes in Pharo, and (unsurprisingly) a `BorderedMorph` is a `Morph` with a

Listing 4-12 Initialize the game

```
initialize
  | sampleCell width height n |
  super initialize.
  n := self cellsPerSide.
  sampleCell := LOCell new.
  width := sampleCell width.
  height := sampleCell height.
  self bounds: (5 @ 5 extent: (width * n) @ (height * n) + (2 * self
    borderWidth)).
  cells := Array2D new: n tabulate: [ :i :j | self newCellAt: i at:
    j ]
```

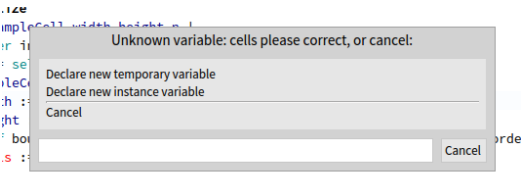

Figure 4-13 Declaring `cells` as a new instance variable.

border. We could also insert the names of the instance variables between the quotes on the second line, but for now, let's just leave that list empty.

4.9 Initializing our game

Now let's define an `initialize` method for `LOGame`. Type the following into the browser as a method for `LOGame` and **Accept** it.

Pharo will complain that it doesn't know the meaning of `cells` (see Figure 4-13). It will offer you a number of ways to fix this.

Choose **Declare new instance variable**, because we want `cells` to be an instance variable.

4.10 Taking advantage of the debugger

At this stage if you open a `Playground`, type `LOGame new`, and **Do it**, Pharo will complain that it doesn't know the meaning of some of the terms (see Figure 4-14). It will tell you that it doesn't know of a message `cellsPerSide`, and will open a debugger. But `cellsPerSide` is not a mistake; it is just a method that we haven't yet defined. We will do so, shortly.

Now let us do it: type `LOGame new` and **Do it**. Do not close the debugger. Click on the button `Create` of the debugger, when prompted, select **LOGame**,

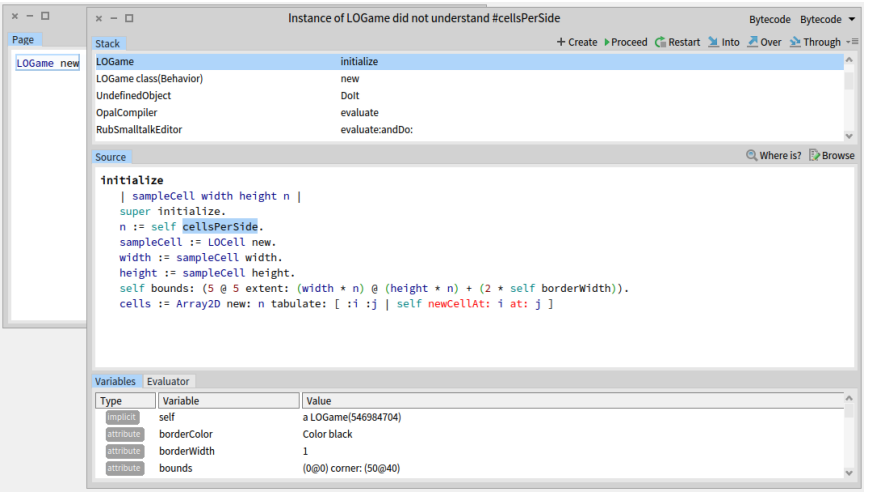

Figure 4-14 Pharo detecting an unknown selector.

the class which will contain the method, click on **ok**, then when prompted for a method protocol enter `accessing`. The debugger will create the method `cellsPerSide` on the fly and invoke it immediately. As there is no magic, the created method will simply raise an exception and the debugger will stop again (as shown in Figure 4-15) giving you the opportunity to define the behavior of the method.

Here you can write your method. This method could hardly be simpler: it answers the constant 10. One advantage of representing constants as methods is that if the program evolves so that the constant then depends on some other features, the method can be changed to calculate this value.

```
cellsPerSide
  "The number of cells along each side of the game"
  ^ 10
```

Define the method `cellsPerSide` in the debugger. Do not forget to compile the method definition by using **Accept**. You should obtain a situation as shown by Figure 4-16. If you press the button **Proceed** the program will continue its execution - here it will stop since we did not define the method `newCellAt:`. We could use the same process but for now we stop to explain a bit what we did so far. Close the debugger, and look at the class definition once again (which you can do by clicking on **LOGame** on the second pane of the **System Browser**), you will see that the browser has modified it to include the instance variable `cells`.

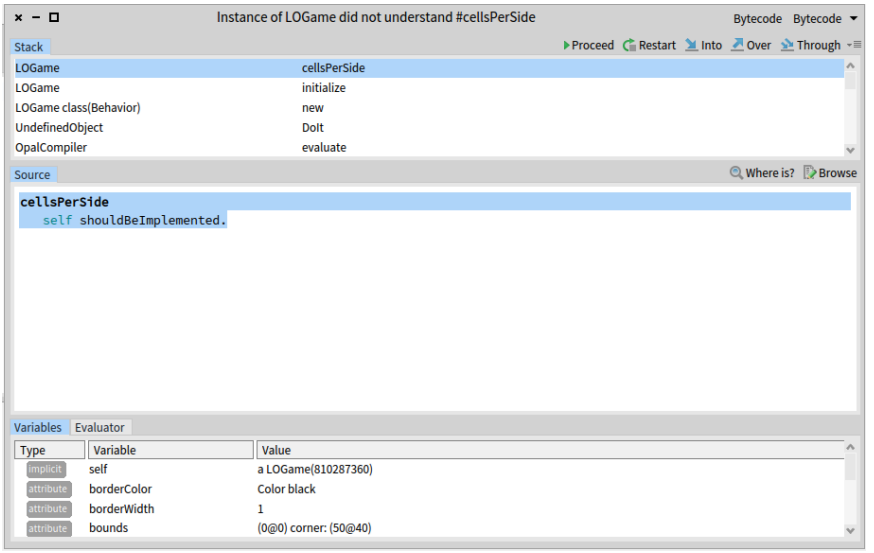

Figure 4-15 The system created a new method with a body to be defined.

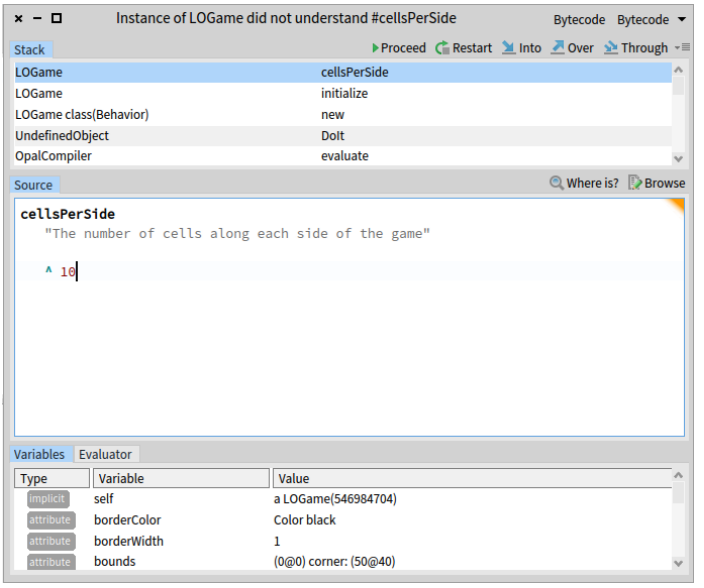

Figure 4-16 Defining `cellsPerSide` in the debugger.

Listing 4-17 Initialize the game

```
initialize
  | sampleCell width height n |
  super initialize.
  n := self cellsPerSide.
  sampleCell := LOCell new.
  width := sampleCell width.
  height := sampleCell height.
  self bounds: (50 @ 50 extent: (width * n) @ (height * n) + (2 *
    self borderWidth)).
  cells := Array2D
    new: n
    tabulate: [ :i :j | self newCellAt: i at: j ]
```

4.11 Studying the initialize method

Let us now study the method `initialize`.

Line 2

At line 2, the expression `| sampleCell width height n |` declares 4 temporary variables. They are called temporary variables because their scope and lifetime are limited to this method. Temporary variables with explanatory names are helpful in making code more readable. Lines 4-7 set the value of these variables.

How big should our game board be? Big enough to hold some integral number of cells, and big enough to draw a border around them. How many cells is the right number? 5? 10? 100? We don't know yet, and if we did, we would probably change our minds later. So we delegate the responsibility for knowing that number to another method, which we name `cellsPerSide`, and which we will write in a minute or two. Don't be put off by this: it is actually good practice to code by referring to other methods that we haven't yet defined. Why? Well, it wasn't until we started writing the `initialize` method that we realized that we needed it. And at that point, we can give it a meaningful name, and move on, without interrupting our flow.

Line 4

The fourth line uses this method, `n := self cellsPerSide.` sends the message `cellsPerSide` to `self`, i.e., to this very object. The response, which will be the number of cells per side of the game board, is assigned to `n`.

The next three lines create a new `LOCell` object, and assign its width and height to the appropriate temporary variables.

Line 8

Line 8 sets the bounds of the new object. Without worrying too much about the details just yet, believe us that the expression in parentheses creates a square with its origin (i.e., its top-left corner) at the point (50,50) and its bottom-right corner far enough away to allow space for the right number of cells.

Last line

The last line sets the `LOGame` object's instance variable `cells` to a newly created `Array2D` with the right number of rows and columns. We do this by sending the message `new: tabulate:` to the `Array2D` class (classes are objects too, so we can send them messages). We know that `new: tabulate:` takes two arguments because it has two colons (:) in its name. The arguments go right after the colons. If you are used to languages that put all of the arguments together inside parentheses, this may seem weird at first. Don't panic, it's only syntax! It turns out to be a very good syntax because the name of the method can be used to explain the roles of the arguments. For example, it is pretty clear that `Array2D rows: 5 columns: 2` has 5 rows and 2 columns, and not 2 rows and 5 columns.

`Array2D new: n tabulate: [ :i :j | self newCellAt: i at: j ]` creates a new n X n two dimensional array (matrix) and initializes its elements. The initial value of each element will depend on its coordinates. The $(i,j)^{th}$ element will be initialized to the result of evaluating `self newCellAt: i at: j`.

4.12 Organizing methods into protocols

Before we define any more methods, let's take a quick look at the third pane at the top of the browser. In the same way that the first pane of the browser lets us categorize classes into packages, the protocol pane lets us categorize methods so that we are not overwhelmed by a very long list of method names in the method pane. These groups of methods are called "protocols".

By default, you will have `instance side` virtual protocol, which contains all of the methods in the class.

If you have followed along with this example, the protocol pane may well contain the `initialization` and `overrides` protocols. These protocols are added automatically when you override `initialize`. Pharo 8 **System Browser** organizes the methods automatically, and add them to the appropriate protocol, when possible.

How does the **System Browser** know that this is the right protocol? Well, in general Pharo can't know exactly, but for example if there is also an `ini-`

tialize method in the superclass, and it assumes that our initialize method should go in the same protocol as the one that it overrides.

The protocol pane may contain the protocol **as yet unclassified**. Methods that aren't organized into protocols can be found here. You can right-click in the protocol pane and select **categorize all uncategorized** to fix this, or you can organize manually.

4.13 A typographic convention

Pharoers frequently use the notation `Class >> method` to identify the class to which a method belongs. For example, the `cellsPerSide` method in class `LOGame` would be referred to as `LOGame >> cellsPerSide`. Just keep in mind that this is not Pharo syntax exactly, but merely a convenient notation to indicate "the instance method `cellsPerSide` which belongs to the class `LOGame`". The corresponding notation for a class-side method would be `LOGame class >> #someClassSideMethod`.

From now on, when we show a method in this book, we will write the name of the method in this form. Of course, when you actually type the code into the browser, you don't have to type the class name or the >>; instead, you just make sure that the appropriate class is selected in the class pane.

4.14 Finishing the game

Now let's define the other method that are used by `LOGame >> initialize`. Let's define `LOGame >> newCellAt: at:` in the `initialization` protocol.

```
LOGame >> newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on-screen
    representation at the appropriate screen position. Answer the
    new cell"

  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
  c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ].
```

Pay attention the previous code is not fully correct. Therefore, it will produce an error and this is on purpose.

Formatting.

As you can see there are some tabulation and empty lines. To keep the same convention you can right-click on the method edit area and click on `Format`

Listing 4-18 The callback method

```
LOGame >> toggleNeighboursOfCellAt: i at: j

  i > 1
    ifTrue: [ (cells at: i - 1 at: j) toggleState ].
  i < self cellsPerSide
    ifTrue: [ (cells at: i + 1 at: j) toggleState ].
  j > 1
    ifTrue: [ (cells at: i at: j - 1) toggleState ].
  j < self cellsPerSide
    ifTrue: [ (cells at: i at: j + 1) toggleState ]
```

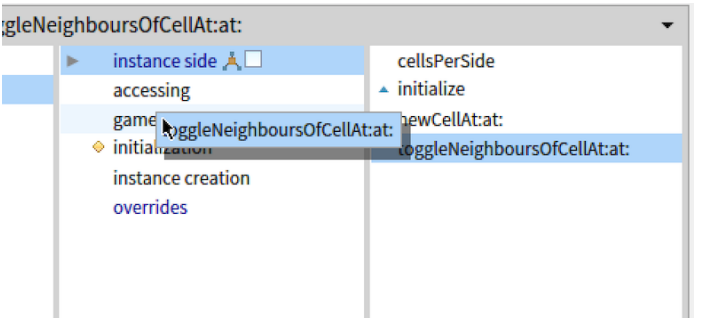

Figure 4-19 Drag a method to a protocol.

(or use `CMD-Shift-f` shortcut). This will format your method.

Toggle neighbours.

The method defined above created a new `LOCell`, initialized to position (i, j) in the Array2D of cells. The last line defines the new cell's `mouseAction` to be the block `[ self toggleNeighboursOfCellAt: i at: j ]`. In effect, this defines the callback behaviour to perform when the mouse is clicked. The corresponding method also needs to be defined.

The method `toggleNeighboursOfCellAt:at:` toggles the state of the four cells to the north, south, west and east of cell (i, j). The only complication is that the board is finite, so we have to make sure that a neighboring cell exists before we toggle its state.

Place this method in a new protocol called `game logic`. (Right-click in the protocol pane to add a new protocol.) To move (re-classify) the method, you can simply click on its name and drag it to the newly-created protocol (see Figure 4-19).

Listing 4-20 A typical setter method

```
LOCell >> mouseAction: aBlock
  mouseAction := aBlock
```

Listing 4-21 An event handler

```
LOCell >> mouseUp: anEvent
  mouseAction value
```

4.15 Final LOCell methods

To complete the Lights Out game, we need to define two more methods in class `LOCell` this time to handle mouse events.

The method above does nothing more than set the cell's `mouseAction` variable to the argument, and then answers the new value. Any method that changes the value of an instance variable in this way is called a *setter method*; a method that answers the current value of an instance variable is called a *getter method*.

Go to the class `LOCell`, define `LOCell >> mouseAction:` and put it in the `accessing` protocol.

Finally, we need to define a method `mouseUp:`. This will be called automatically by the GUI framework if the mouse button is released while the cursor is over this cell on the screen. Add the `LOCell >> mouseUp:` method.

What this method does is to send the message `value` to the object stored in the instance variable `mouseAction`. In `LOGame >> newCellAt: i at: j` we created the *block* `[self toggleNeighboursOfCellAt: i at: j ]` which is toggling all the neighbours of a cell and we assigned this block to the `mouseAction` of the cell. Therefore sending the `value` message causes this block to be evaluated, and consequently the state of the cells will toggle.

4.16 Using the debugger

That's it: the Lights Out game is complete! If you have followed all of the steps, you should be able to play the game, consisting of just 2 classes and 7 methods. In a Playground, type `LOGame new openInHand` and **Do it** .

The game will open, and you should be able to click on the cells and see how it works. Well, so much for theory... When you click on a cell, a debugger will appear. In the upper part of the debugger window you can see the execution stack, showing all the active methods. Selecting any one of them will show, in the middle pane, the code being executed in that method, with the part that triggered the error highlighted.

Click on the line labeled `LOGame >> toggleNeighboursOfCellAt: at:` (near the top). The debugger will show you the execution context within this

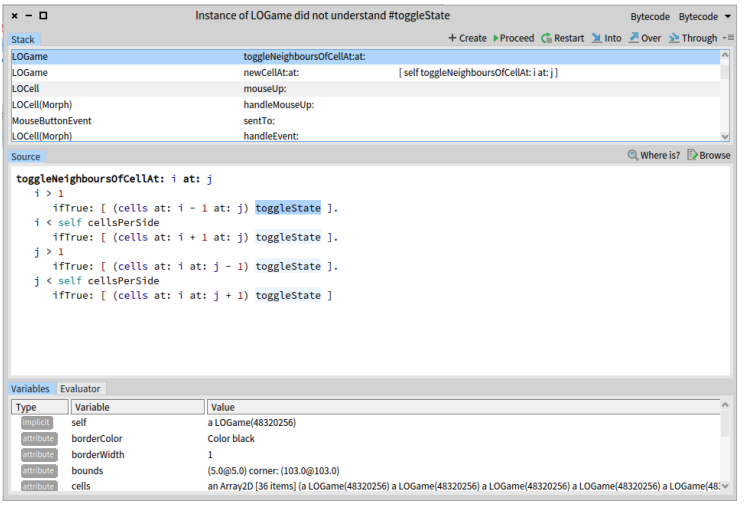

Figure 4-22 The debugger, with the method `toggleNeighboursOfCell:at:` selected.

method where the error occurred (see Figure 4-22).

At the bottom of the debugger is a variable zone. You can inspect the object that is the receiver of the message that caused the selected method to execute, so you can look here to see the values of the instance variables. You can also see the values of the method arguments.

Using the debugger, you can execute code step by step, inspect objects in parameters and local variables, evaluate code just as you can in a playground, and, most surprisingly to those used to other debuggers, change the code while it is being debugged! Some Pharoers program in the debugger almost all the time, rather than in the browser. The advantage of this is that you see the method that you are writing as it will be executed, with real parameters in the actual execution context.

In this case we can see in the first line of the top panel that the `toggleState` message has been sent to an instance of `LOGame`, while it should clearly have been an instance of `LOCell`. The problem is most likely with the initialization of the cells matrix. Browsing the code of `LOGame >> initialize` shows that `cells` is filled with the return values of `newCellAt: at:`, but when we look at that method, we see that there is no return statement there! By default, a method returns `self`, which in the case of `newCellAt: at:` is indeed an instance of `LOGame`. The syntax to return a value from a method in Pharo is `^`.

Close the debugger window. Add the expression `^ c` to the end of the method `LOGame >> newCellAt:at:` so that it returns `c`.

Listing 4-23 Fixing the bug.

```
LOGame >> newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on-screen
    representation at the appropriate screen position. Answer the
    new cell"

  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
  c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ].
  ^ c
```

Listing 4-24 Overriding mouse move actions

```
LOCell >> mouseMove: anEvent
```

Often, you can fix the code directly in the debugger window and click `Proceed` to continue running the application. In our case, because the bug was in the initialization of an object, rather than in the method that failed, the easiest thing to do is to close the debugger window, destroy the running instance of the game (with the halo `CMD-Alt-Shift` and click), and create a new one.

Execute `LOGame new openInHand` again because if you use the old game instance it will still contain the block with the old logic.

Now the game should work properly... or nearly so. If we happen to move the mouse between clicking and releasing, then the cell the mouse is over will also be toggled. This turns out to be behavior that we inherit from `SimpleSwitchMorph`. We can fix this simply by overriding `mouseMove:` to do nothing:

Finally we are done!

About the debugger.

By default when an error occurs in Pharo, the system displays a debugger. However, we can fully control this behavior. For example we can write the error in a file. We can even serialize the execution stack in a file, zip and reopen it in another image. Now when we are in development mode the debugger is available to let us go as fast as possible. In production system, developers often control the debugger to hide their mistakes from their clients.

4.17 In case everything fails

First do not stress! It is normal to mess up and there is no point to have Second if you do not succeed to delete the game. Try to get an inspector on any graphical element of the game using the halos: Option-Shift+Click and choose menu and the debug... menu and inspect Morph.

From there you can execute

- if you are inspecting the game itself: `self delete`.
- if you are inspect a game cell: `self owner delete`.

4.18 Saving and sharing Pharo code

Now that you have **Lights Out** working, you probably want to save it somewhere so that you can archive it and share it with your friends. Of course, you can save your whole Pharo image, and show off your first program by running it, but your friends probably have their own code in their images, and don't want to give that up to use your image. What you need is a way of getting source code out of your Pharo image so that other programmers can bring it into theirs.

We'll discuss the various ways to save and share code in a subsequent chapter, Chapter 5. For now, here is an overview of some of the available methods.

4.19 Iceberg: Pharo and Git

Iceberg is the new default tool for versioning your code using git and handling git repositories directly from Pharo images.

Declared repositories.

Iceberg is accessible through world menu `Tools > Iceberg`. When opened you will first see a **Repositories** screen. There you can find all git repositories managed by Iceberg. Do not care about the Pharo repository, it is there for people that want to contribute to Pharo.

Adding a new repository.

To manage our example with git and Iceberg, we should add it first. Press Add on the toolbar of **Repositories** screen. On the Figure 4-26 there are multiple options. The one we are interested in is **New repository**. Enter the project name like in the Figure 4-26, and if you wish you can leave the src blank. You can name your project PBE-LightsOut.

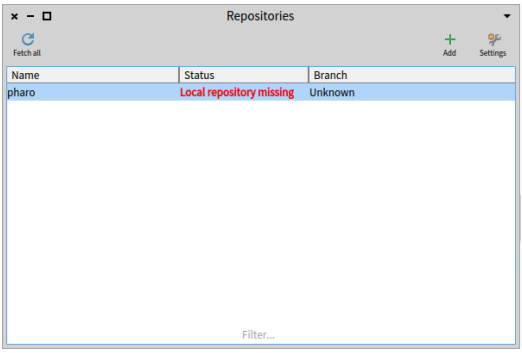

Figure 4-25 Repositories screen.

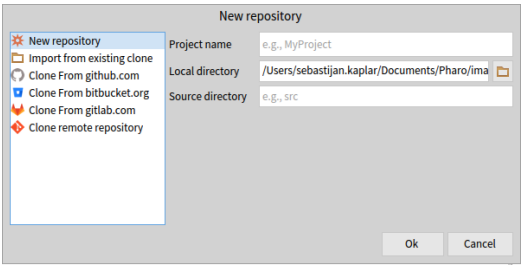

Figure 4-26 Creating new repository.

Adding a package.

After creating new repository you will notice that the **Repositories** screen has changed. It will contain newly created repository, with the ”Not Loaded” status, which basically means, that now your repository is empty and that you should add packages to it (see Figure 4-27).

Now, to add packages, double click on the PBE-LightsOut repository, and in the new dialog press `Add package` in the toolbar. Select checkmark before the **PBE-LightsOut** package and press `Add`, it is also possible to add multiple packages by selecting checkmark before the package name. The repository now contains the added package, with the status **Uncommitted changes**, as shown in Figure 4-28.

Committing changes.

One more thing remains is to actually commit those changes. Select `Commit` on the toolbar to get the commit dialog. Here you can see all the files that are beeing committed, with a green plus next to their name (Figure 4-29). Before committing enter the meaningful message for that particular commit,

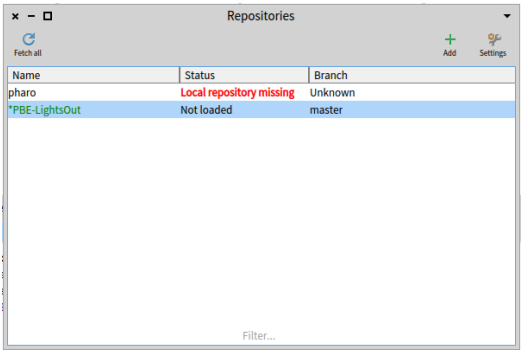

Figure 4-27 Updated repositories screen.

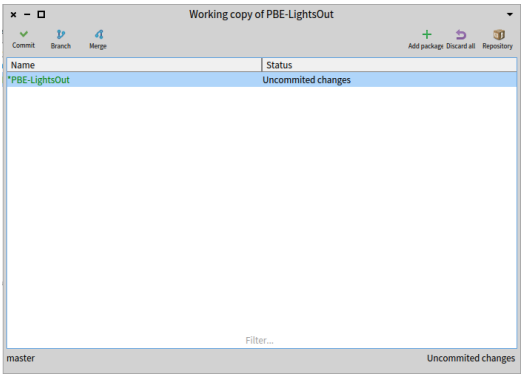

Figure 4-28 Iceberg working copy dialog.

and finally press `Commit`. Note that Iceberg adds some metadata about the file format and the location of the code without the directory.

In this example, we used Iceberg to version our project using git. Have in mind that all committed changes, are performed on your *local* machine. To connect your local repository, to remote repository (e.g., GitHub) you need to add it, you can do that by selecting `Repository` on the Working copy dialog browser (Figure 4-28) and select the **Add remote** button on the top right. This is covered in Chapter 5.

4.20 Saving code in a file

If you do not want to use a version control system such Git you can save the code of a package, class, or method simply.

You can also save the code of your package by ”filing out” the code. The

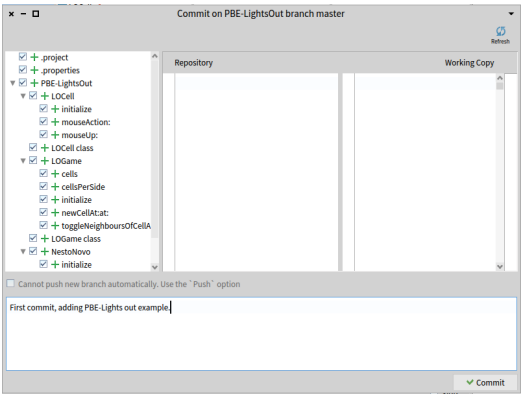

Figure 4-29 Iceberg working copy dialog.

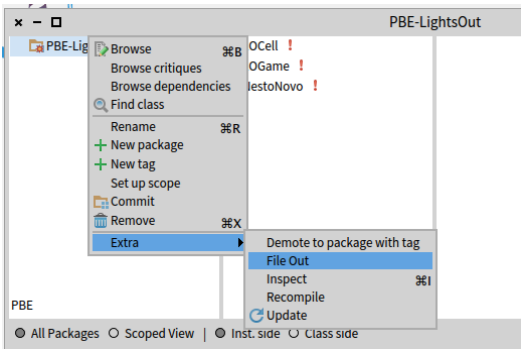

Figure 4-30 File Out our PBE-LightsOut.

right-click menu in the Package pane will give you the option to `Extra > File Out` the whole of package `PBE-LightsOut`. The resulting file is more or less human readable, but is really intended for computers, not humans. You can email this file to your friends, and they can file it into their own Pharo images using the file list browser.

Right-click on the `PBE-LightsOut` package and file out the contents (see Figure 4-30). You should now find a file named `PBE-LightsOut.st` in the same folder on disk where your image is saved. Have a look at this file with a text editor.

Open a fresh Pharo image and use the `File Browser` tool (`Tools --> File Browser`) to file in the `PBE-LightsOut.st` fileout (see Figure 4-31) and `fileIn`. Verify that the game now works in the new image.

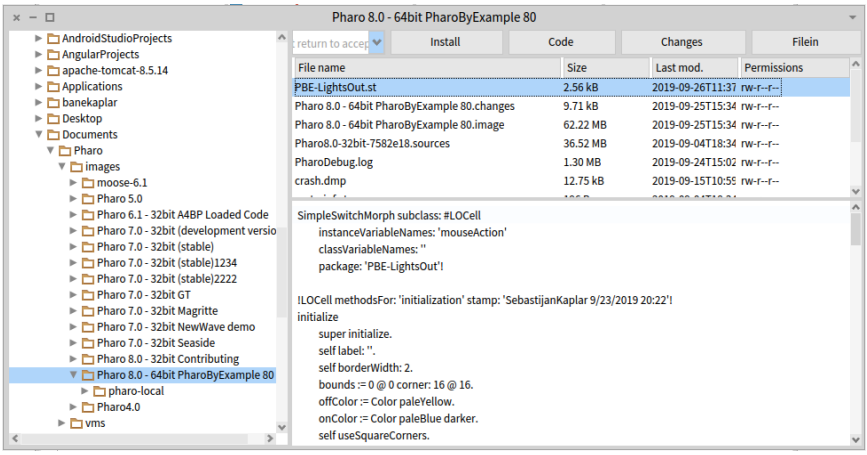

Figure 4-31 Import your code with the file browser.

4.21 About Setter/Getter convention

If you are used to getters and setters in other programming languages, you might expect these methods to be called `setMouseAction` and `getMouseAction`. The Pharo convention is different. A getter always has the same name as the variable it gets, and a setter is named similarly, but with a trailing ":", hence `mouseAction` and `mouseAction:`. Collectively, setters and getters are called *accessor methods*, and by convention they should be placed in the `accessing` protocol. In Pharo, all instance variables are private to the object that owns them, so the only way for another object to read or write those variables is through accessor methods like this one. In fact, the instance variables can be accessed in subclasses too.

4.22 On categories vs. packages

Historically, Pharo packages were implemented as "categories" (a group of classes). With the newer versions of Pharo, the term category is being deprecated, and replaced exclusively by package.

If you use an older version of Pharo or an old tutorial, the class template will be as follow:

```
SimpleSwitchMorph subclass: #LOCell
  instanceVariableNames: 'mouseAction'
  classVariableNames: ''
  category: 'PBE-LightsOut'
```

It is equivalent to the one we mentioned earlier. In this book we only use the term **package**. The Pharo package is also what you will be using to ver-

sion your source code using Iceberg versioning tool: the new tool to manage source code via Git. This book version has two new chapters covering Iceberg and package management (see Chapters 5 and **??**).

4.23 Chapter summary

In this chapter you have seen how to create packages, classes and methods. In addition, you have learned how to use the System browser, the inspector, the debugger and Iceberg to version your code using git.

- Packages are groups of related classes.
- A new class is created by sending a message to its superclass.
- Protocols are groups of related methods inside a class.
- A new method is created or modified by editing its definition in the browser and then accepting the changes.
- The inspector offers a simple, general-purpose GUI for inspecting and interacting with arbitrary objects.
- The browser detects usage of undeclared variables, and offers possible corrections.
- The `initialize` method is automatically executed after an object is created in Pharo. You can put any initialization code there.
- The debugger provides a high-level GUI to inspect and modify the state of a running program.
- You can share source code by filing out a package, class or method.
- Using Iceberg and git.

CHAPTER 5

Publishing your first Pharo project

In this chapter we explain how you can publish your project on Github using Iceberg. We do not explain basic concepts like commit, push/pull, merging, or cloning.

A strong precondition before reading this chapter is that you must be able to publish from the command line to the `git` hosting service that you want to use. If you cannot do not expect Iceberg to fix it magically for you. Now if you have some problems with SSH configuration (which is the default with Github) you can either use HTTPS or have a look in *Manage your code with Iceberg* booklet. Let us get started.

5.1 For the impatient

If you do not want to read everything, here is the executive summary.

- Create a project on Github or any git-based platform.
- [Optional] Configure Iceberg to use custom ssh keys.
- Add a project in Iceberg.
  - Optionally but strongly recommanded, in the cloned repository, create a directory named `src` on your file system. This is a good convention.
- In Iceberg, open your project and add your packages.
- Commit your project.

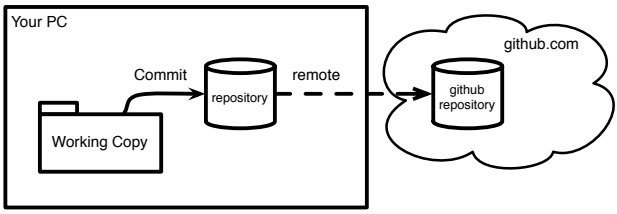

Figure 5-1 A distributed versioning system.

- [Optional] Add a baseline to ease loading your project.
- Push your change to your remote repository.

You are done. Now we can explain calmly.

5.2 Basic Architecture

As `git` is a distributed versioning system, you need a *local* clone of the repository and a *working copy*. Your working copy and local repository are usually on your machine. This is to this local repository that your changes will be commited to before being pushed to remote repositories (Figure 5-1). We will see in the next Chapter that the situation is a bit more complex and that Iceberg is hiding the extra complexity for us.

5.3 Create a new project on Github

While you can save locally first and then later create a remote repository, in this chapter we first create a new project on Github. Figure 5-2 shows the creation of a project on Github. The order does not really matter. What is different is that you should use different options when add a repository to Iceberg as we will show later.

5.4 [Optional] SSH setup: Tell Iceberg to use your keys

To be able to commit to your `git` project, you should either use HTTPS or you will need to set up valid credentials in your system. In case you use SSH (the default way), you will need to make sure those keys are available to your Github account and also that the shell adds them for smoother communication with the server.s

Go to settings browser, search for "Use custom SSH keys" and enter your data there as shown in Figure 5-3).

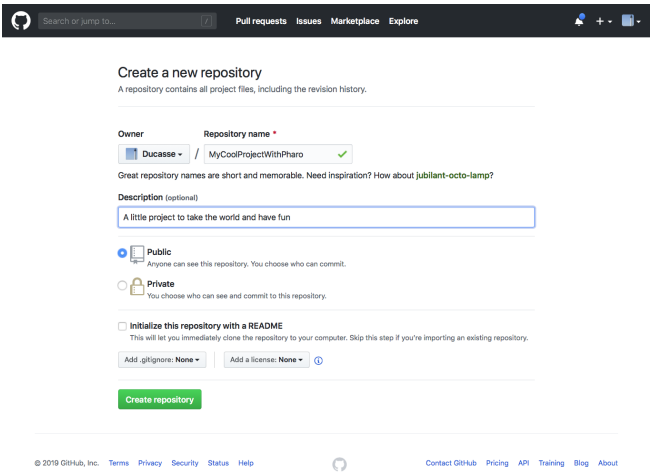

Figure 5-2 Create a new project on Github.

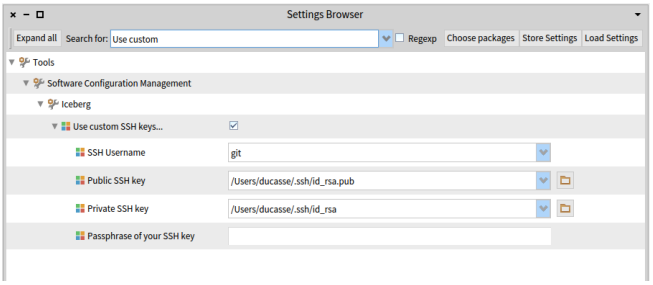

Figure 5-3 Use Custom SSH keys settings.

Alternatively, you can execute the following expressions in your image playground or add them to your Pharo system preference file (See Menu System item startup):

```
IceCredentialsProvider useCustomSsh: true.
IceCredentialsProvider sshCredentials
  publicKey: 'path\to\ssh\id_rsa.pub';
  privateKey: 'path\to\ssh\id_rsa'
```

Note **Pro Tip:** this can be used too in case you have a non-default key file. You just need to replace `id_rsa` with your file name.

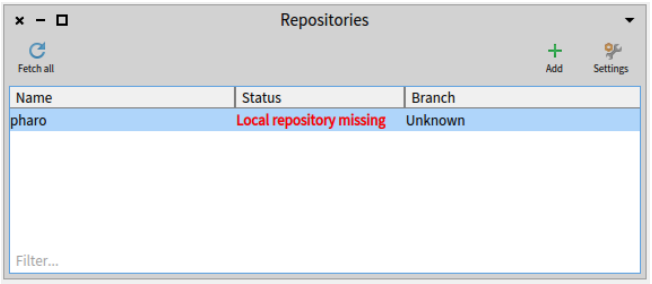

Figure 5-4 Iceberg *Repositories* browser on a fresh image indicates that if you want to version modifications to Pharo itself you will have to tell Iceberg where the Pharo clone is located. But you do not care.

5.5 Iceberg *Repositories* browser

Figure 5-4 shows the top level Iceberg pane. It shows that for now you do not have defined nor loaded any project. It shows the Pharo project and indicates that it could not find its local repository by displaying 'Local repository missing'.

First you do not have to worry about the Pharo project or repository if you do not want to contribute to Pharo. So just go ahead. Now if you want to understand what is happening here is the explanation. The Pharo system does not have any idea where it should look for the `git` repository corresponding to the source of the classes it contains. Indeed, the image you are executing may have been built somewhere, patched or not many times. Now Pharo is fully operational without having a local repository. You can browse system classes and methods because Pharo has its own internal source management. This warning just indicates that if you want to version Pharo system code using `git` then you should indicate to the system where the clone and working copy are located on your local machine. So if you do not plan to modify and version the Pharo system code, you do not have to worry.

5.6 Add a new project to Iceberg

The first step is then to add a project to Iceberg:

- Press the '+' button to the right of the Iceberg main window.
- Select the source of your project. In our example, since you did not clone your project yet, choose the Github option.

Notice that you can either use SSH (Figure 5-5) or HTTPS (Figure 5-6).

Figure 5-5 and 5-6) instruct Iceberg to clone the repository we just created on Github. We specify the owner, project, and physical location where the local

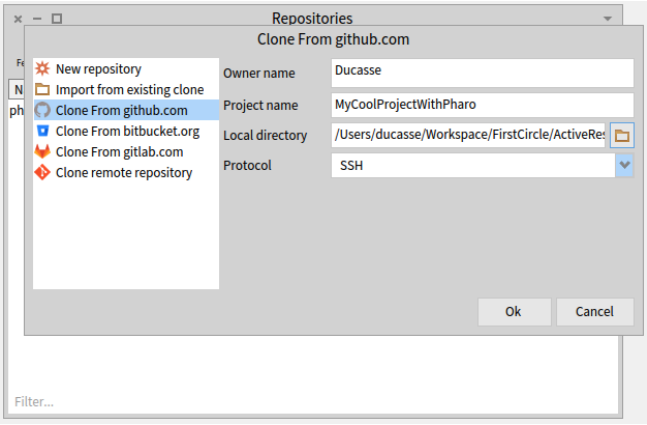

Figure 5-5 Cloning a project hosted on Github via SSH.

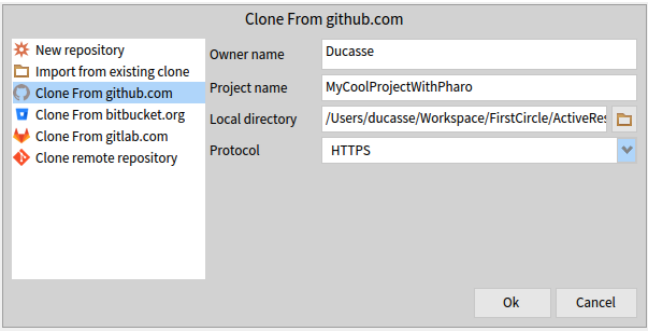

Figure 5-6 Cloning a project hosted on Github via HTTPS.

clone and `git` working copy will be on your disk.

Iceberg has now added your project to its list of managed projects and cloned an empty repository to your disk. You will see the status of your project, as in Figure 5-7. Here is a breakdown of what you are seeing:

- MyCoolProjectWithPharo has a star and is green. This usually means that you have changes which haven't been committed yet, but may also happen in unrelated edge cases like this one. Don't worry about this for now.
- The Status of the project is 'No Project Found' and this is more important. This is normal since the project is empty. Iceberg cannot find its metadata. We will fix this soon.

Later on, when you will have commited changes to your project and you want to load it in another image, when you will clone again, you will see that

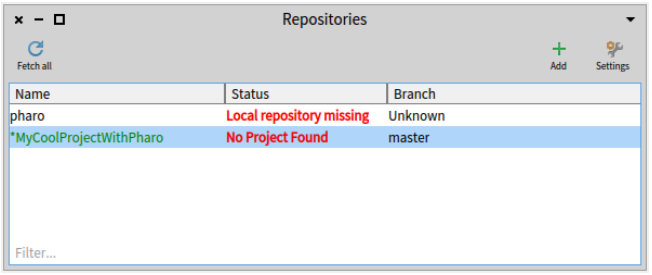

Figure 5-7 Just after cloning an empty project, Iceberg reports that the project is missing information.

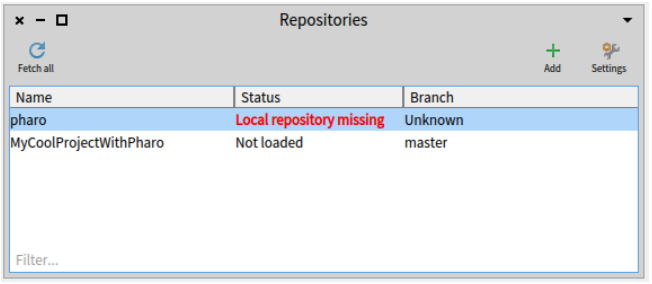

Figure 5-8 Adding a project with some contents shows that the project is not loaded - not that it is not found.

Iceberg will just report that the project is not loaded as shown in Figure 5-8.

5.7 Repair to the rescue

Iceberg is a smart tool that tries to help you fix the problems you may encounter while working with `git`. As a general principle, each time you get a status with red text (such as "No Project Found" or "Detached Working Copy"), you should ask Iceberg to fix it using the **Repair** command.

Iceberg cannot solve all situations automatically, but it will propose and explain possible repair actions. The actions are ranked from most to least likely to be right one. Each action has a displayed explanation on the situation and the consequences of the using it. It is always a good idea to read them. Setting your repository the right way makes it extremely hard to lose any piece of code with Iceberg and Pharo is general since Pharo contains its own copy of the code.

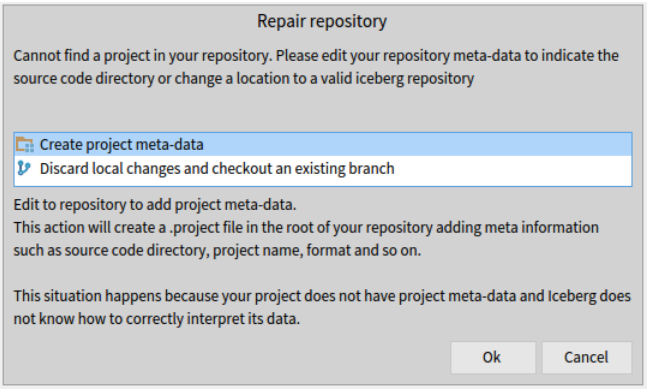

Figure 5-9 Create project metadata action and explanation.

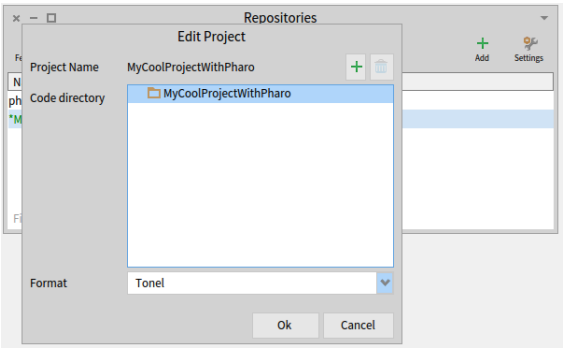

Figure 5-10 Showing where the metadata will be saved and the format encodings.

5.8 Create project metadata

Iceberg reported that it could not find the project because some meta data were missing such as the format of the code encodings and the example location inside the repository. When we activate the repair command we get Figure 5-9. It shows the ”Create project metadata” action and its explanation.

When you choose to create the project metadata, Iceberg shows you the filesystem of your project as well as the repository format as shown in Figure 5-10. Tonel is the preferred format for Pharo projects. It has been designed to be Windows and file system friendly. Change it only if you know what you are doing!

Before accepting the changes, it is a good idea to add a source (`src`) folder

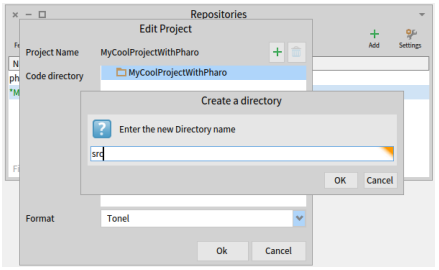

Figure 5-11 Adding a src repository for code storage.

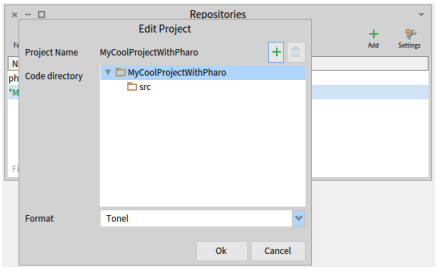

Figure 5-12 Resulting situation with a src folder.

to your repository. Do that by pressing the + icon. You will be prompted to specify the folder for code as shown in Figure 5-11. Iceberg will show you the exact structure of your project as shown in Figure 5-12.

After accepting the project details, Iceberg shows you the files that you will be committing as shown in Figure 5-13

Once you have committed the metadata, Iceberg shows you that your project has been repaired but is not loaded as shown in Figure 5-8. This is normal since we haven't added any packages to our project yet. You can optionally push your changes to your remote repository.

Your local repository is ready, let's move on to the next part.

5.9 Add and commit your package using the *Working copy* browser

Once your project contains Iceberg metadata, Iceberg will be able to manage it easily. Double click on your project to bring a *Working copy* browser for your project. It lists all the packages that compose your project. Right now you have none. Add a package by pressing the + (Add Package) iconic button as shown by Figure 5-14.

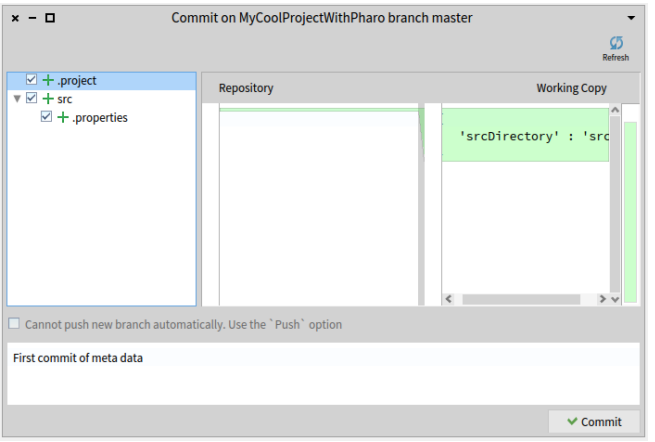

Figure 5-13 Details of metadata commit.

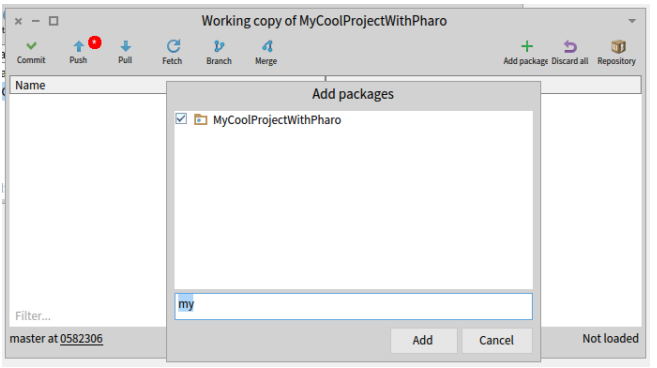

Figure 5-14 Adding a package to your project using the *Working copy* browser.

Again, Iceberg shows that your package contains changes that are not committed using the green color and the star in front of the package name as showing in Figure 5-15.

Commit the changes

Commit the changes to your local repository using the Commit button as shown in Figure 5-16. Iceberg lets you chose the changed entities you want to commit. Here this is not needed but this is an important feature. Iceberg will show the result of the commit action by removing the star and changing the color. It now shows that the code in the image is in sync with your local repository as shown by Figure 5-17. You can commit several times if needed.

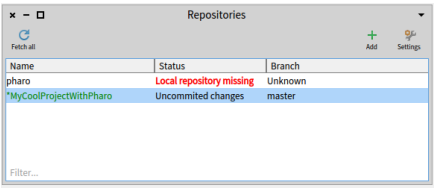

Figure 5-15 Iceberg indicates that your package has unsaved changes – indeed you just added your package.

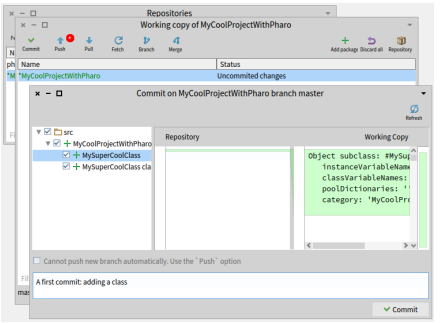

Figure 5-16 When you commit changes, Iceberg shows you the code about to be committed and you can chose the code entities that will effectively be saved.

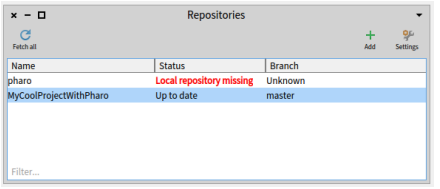

Figure 5-17 Once changes committed, Iceberg reflects that your project is in sync with the code in your local repository.

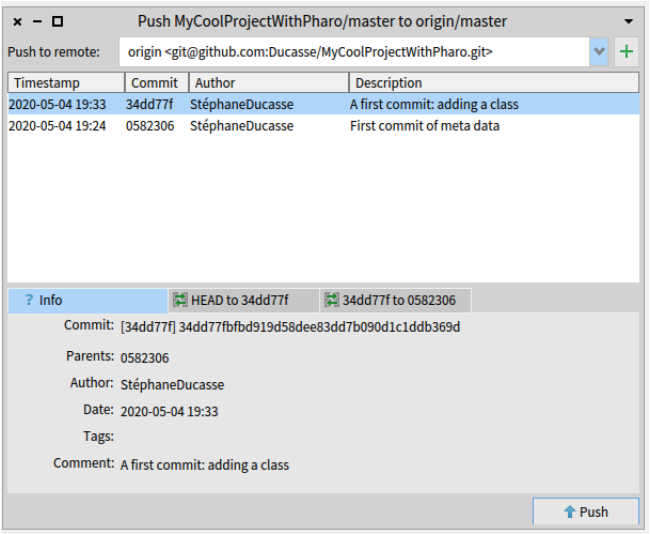

Figure 5-18 Publishing your committed changes.

Publish your changes to your remote

Now you are nearly done. Publish your changes from your local directory to your remote repository using the Push button. You may be prompted for credentials if you used HTTPS.

When you push your changes, Iceberg will show you all the commits awaiting publication and will push them to your remote repository as shown in Figure 5-18. The figure shows the commits we are about to make to add a baseline, which will allow you to easily load your project in other images.

Now you are basically done.

5.10 Conclusion

You now know the essential aspects of managing your code with Github. Iceberg has been designed to guide you so please listen to it unless you really know what you are doing. You are now ready to use services offered around Github to improve your code control and quality!

CHAPTER 6

Configure your project nicely

Versioning code is just the first part of making sure that you and others can reload your code. In this chapter we describe how to define a baseline, a project map that you will use to define dependencies within your project and dependencies to other projects. We also show how to add a good `.gitignore` file. In the next chapter we will show how to configure your project to get more out of the services offered within the Github ecosystem such as Travis-ci to execute automatically your tests.

We start by showing you how you can commit your code if you did not create your remote repository first.

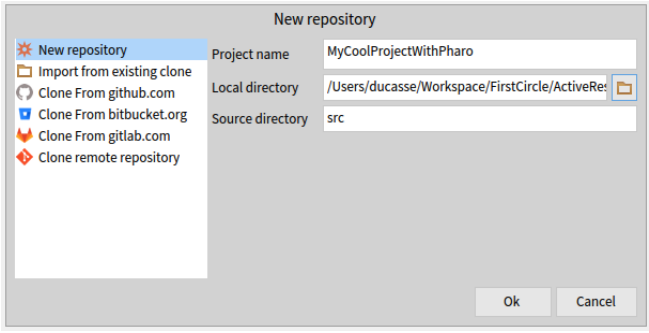

Figure 6-1 Creating a local repository without pre-existing remote repository.

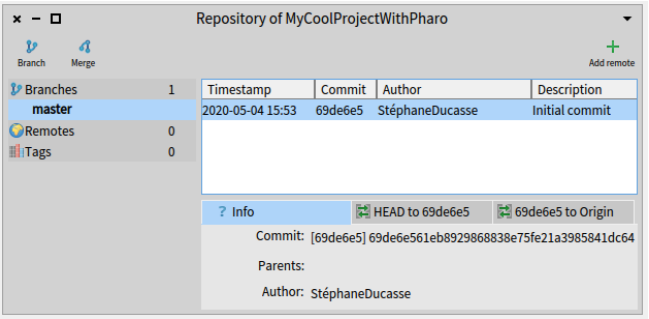

Figure 6-2 Opening the repository browser let you add and browse branches as well as remote repositories.

6.1 What if I did not create a remote repository

In the previous chapter we started by creating a remote repository on Github. Then we asked Iceberg to add a project by cloning it from Github. Now you may ask yourself what is the process to publish first your project locally without a pre-existing repository. This is actually simple.

Create a new repository.

When you add a new repository use the 'New repository' option as shown in 6-1.

Add a remote.

If you want to commit to a remote repository, you will have to add it using the *Repository* browser. You can access this browser through the associated menu item or the icon. The *Repository* browser gives you access to the `git` repositories associated with your project: you can access, manage branches and also add or remove remote repositories. Figure 6-3 shows the repository browser on our project.

Pressing on the 'Add remote' iconic button adds a remote by filling the needed information that you can find in your Github project. Figure 6-3 shows it for the sample project using SSH and Figure 6-4 for HTTPS.

Push to the remote.

Now you can push your changes and versions to the remote repository using the Push iconic button. Once you have pushed you can see that you have one remote as shown in Figure 6-5.

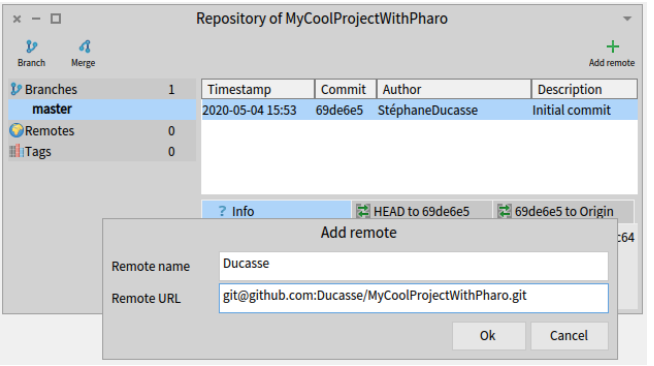

Figure 6-3 Adding a remote using the *Repository* browser of your project (SSH version).

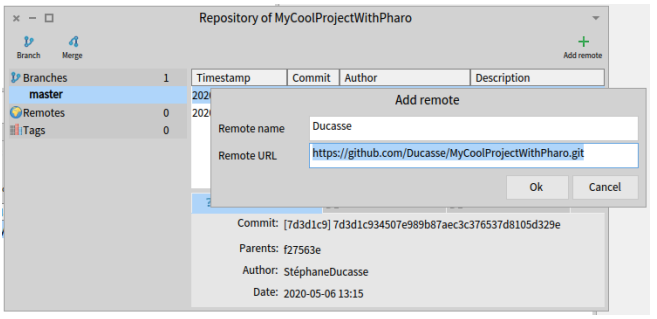

Figure 6-4 Adding a remote using the *Repository* browser of your project (HTTP version).

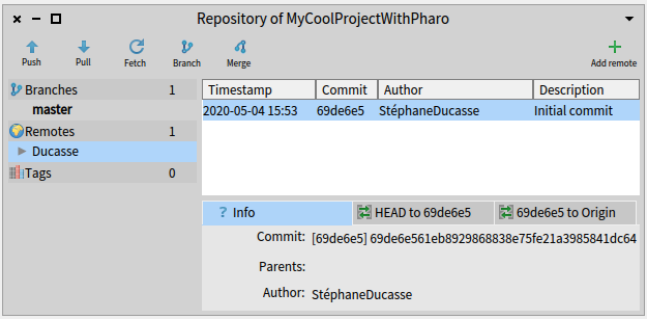

Figure 6-5 Once you pushed you changes to the remote repository.

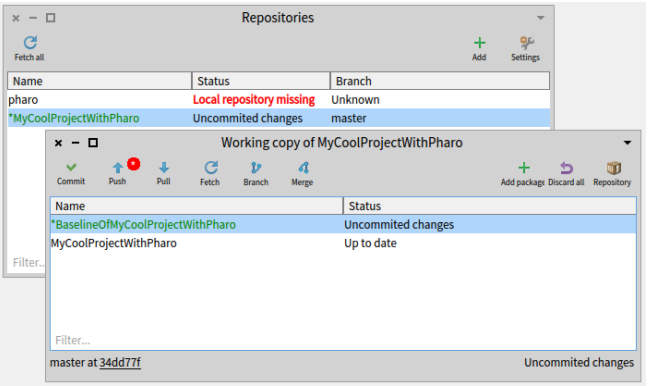

Figure 6-6 Added the baseline package to your project using the *Working copy* browser.

6.2 Defining a `BaselineOf`

A *baseline* is a description of the architecture of a project. You will express the dependencies between your packages and other projects so that all the dependent projects are loaded without the user having to understand them or the links between them.

A baseline is expressed as a subclass of `BaselineOf` and packaged in a package named `'BaselineOfXXX'` (where 'XXX' is the name of your project). So if you have no dependencies, you can have something like this.

```
BaselineOf subclass: #BaselineOfMyCoolProjectWithPharo
  ...
  package: 'BaselineOfMyCoolProjectWithPharo'
```

```
BaselineOfMyCoolProjectWithPharo >> baseline: spec
  <baseline>
  spec
    for: #common
    do: [ spec package: 'MyCoolProjectWithPharo' ]
```

Once you have defined your baseline, you should add its package to your project using the working copy browser as explained in the previous chapter. You should obtain the following situation shown in Figure 6-6. Now, commit it and push your changes to your remote repository.

A more elaborated web resources about baseline possibility is available at: https://github.com/pharo-open-documentation/pharo-wiki/.

6.3 Loading from an existing repository

Once you have a repository you committed code to and would like to load it into a new Pharo image, there are two ways to work this out.

Manual load.

- Add the project as explained in the first chapter
- Open the working copy browser by double clicking on the project line in the repositories browser.
- Select a package and manually load it.

Scripting the load.

The second way is to make use of Metacello. However, this will only work if you have already created a `BaselineOf`. In this case, you can just execute the following snippet:

```
Metacello new
  baseline: 'MyCoolProjectWithPharo';
  repository: 'github://Ducasse/MyCoolProjectWithPharo/src';
  load
```

For projects with metadata, like the one we just created, that's it. Notice that we not only mention the Github pass but also added the code folder (here `src`).

6.4 [Optional] Add a nice .gitignore file

Iceberg automatically manages files such as .gitignore.

```
# For Pharo 70 and up
# http://www.pharo.org
# Since Pharo 70 all the community is moving to git.

# image, changes and sources
*.changes
*.sources
*.image

# Pharo Debug log file and launcher metadata
PharoDebug.log
pharo.version
meta-inf.ston

# Since Pharo 70, all local cache files for Monticello package
    cache, playground, epicea... are under the pharo-local
```

```
/pharo-local

# Metacello-github cache
/github-cache
github-*.zip
```

6.5 Going further: Understanding the architecture

As `git` is a distributed versioning system, you need a local clone of your repository. In general you edit your working copy located on your hard-drive and you commit to your local clone, and from there you push to remote repositories like Github. We explain here the specificity of managing Pharo with git.

When coding in Pharo, you should understand that you are not directly editing your local working copy, you are modifying objects that represent classes and methods that are living in the Pharo environment. Therefore it is like you have a double working copy: Pharo itself and the `git` working copy.

When you use `git` command lines, you have to understand that there is the code in the image and the code in the working copy (and your local clone).To update your image, you *first* have to update your `git` working copy and *then* load code from the working copy to the image. To save your code you have to save the code to files, add them to your git working copy and commit them to your clone.

Now the interesting part is that Iceberg manages all this for you transparently. All the synchronization between these two working copies is done behind the scene.

Figure 6-7 shows the architecture of the system.

- You have your code in the Pharo image.
- Pharo is acting as a working copy (it contains the contents of the local `git` repository).
- Iceberg manages the publication of your code to the `git` working copy and the `git` local repository.
- Iceberg manages the publication of your code to remote repositories.
- Iceberg manages the re-synchronization of your image with the `git` local repository, `git` remote repositories and the `git` working copy.

6.6 Conclusion

We show how to package your code correclty. It will help you to reload it.

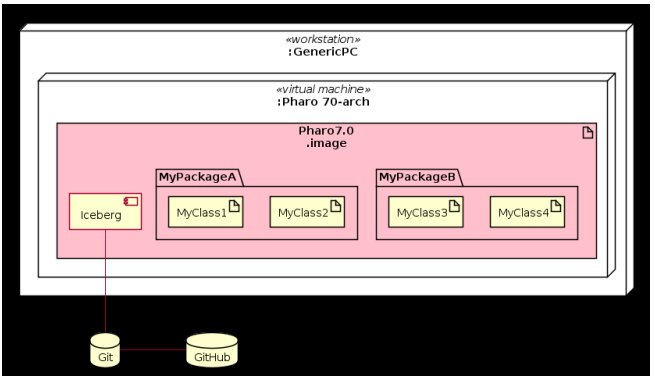

Figure 6-7 Architecture.

CHAPTER 7

Syntax in a nutshell

Pharo adopts a syntax very close to that of its ancestor, Smalltalk. The syntax is designed so that program text can be read aloud as though it were a kind of pidgin English. The following method of the class `Week` shows an example of the syntax. It checks whether `DayNames` already contains the argument, i.e., if this argument represents a correct day name. If this is the case, it will assign it to the class variable `StartDay`.

```
startDay: aSymbol

   (DayNames includes: aSymbol)
      ifTrue: [ StartDay := aSymbol ]
      ifFalse: [ self error: aSymbol, ' is not a recognised day
    name' ]
```

Pharo's syntax is minimal. Essentially there is syntax only for sending messages (i.e., expressions). Expressions are built up from a very small number of primitive elements (message sends, assignments, closures, returns...). There are only 6 reserved keywords, i.e., pseudo-variables, and there are no dedicated syntax constructs for control structures or declaring new classes. Instead, nearly everything is achieved by sending messages to objects. For instance, instead of an if-then-else control structure, conditionals are expressed as messages (such as `ifTrue:`) sent to Boolean objects. New subclasses are created by sending a message to their superclass.

7.1 Syntactic elements

Expressions are composed of the following building blocks:

1. The six *pseudo-variables*: `self`, `super`, `nil`, `true`, `false`, and `thisContext`
2. Constant expressions for *literal* objects including numbers, characters, strings, symbols and arrays
3. Variable declarations
4. Assignments
5. Block closures
6. Messages
7. Method returns

We can see examples of the various syntactic elements in the table below.

Syntax expression	What it represents
`startPoint`	a variable name
`Transcript`	a global variable name
self	pseudo-variable
`1`	decimal integer
`2r101`	binary integer
`1.5`	floating point number
`2.4e7`	number in exponential notation
`$a`	the character `'a'`
`'Hello'`	the string `'Hello'`
`#Hello`	the symbol `#Hello`
`#(1 2 3)`	a literal array
`{ 1 . 2 . 1 + 2 }`	a dynamic array
`"a comment"`	a comment
`| x y |`	declaration of variables `x` and `y`
`x := 1`	assign 1 to `x`
`[:x | x + 2 ]`	a block that evaluates to `x + 2`
`<primitive: 1>`	a method annotation (here primitive)
`3 factorial`	unary message `factorial`
`3 + 4`	binary message `+`
`2 raisedTo: 6 modulo: 10`	keyword message `raisedTo:modulo:`
`^ true`	return the value `true`
`x := 2 . x := x + x`	two expressions separated by separator (`.`)
`Transcript show: 'hello'; cr`	two cascade messages separated by (`;`)

Local variables. `startPoint` is a variable name, or identifier. By convention, identifiers are composed of words in "camelCase" (i.e., each word except the first starting with an upper case letter). The first letter of an instance variable, method or block parameters, or temporary variable must be lower case. This indicates to the reader that the variable has a private scope.

Shared variables. Identifiers that start with upper case letters are global variables, class variables, pool dictionaries or class names. `Transcript` is a global variable, an instance of the class `ThreadSafeTranscript`.

The current object. `self` is a pseudo-variable that refers to the object that receives the message (that led to the execution of the method using `self`). It gives us a way send messages to it. We call `self` "the receiver" because this object will receive the message that causes the method to be executed. Finally, `self` is called a "pseudo-variable" since we cannot directly change its values or assign to it.

Integers. In addition to ordinary decimal integers like 42, Pharo also provides a radix notation. `2r101` is 101 in radix 2 (i.e., binary), which is equal to decimal 5.

Floating point numbers. Such numbers can be specified with their base-ten exponent: `2.4e7` is `2.4 x 10^7`.

Characters. A dollar sign introduces a literal character: `$a` is the literal for the character `'a'`. Instances of special, non-printing characters can be obtained by sending appropriately named messages to the `Character` class, such as `Character space` and `Character tab`.

Strings. Single quotes `' '` are used to define a literal string. If you want a string with a single quote inside, just double the quote, as in `'G''day'`.

Symbols. Symbols are like Strings, in that they contain a sequence of characters. However, unlike a string, a literal symbol is guaranteed to be globally unique. There is only one `Symbol` object `#Hello` but there may be multiple `String` objects with the value `'Hello'`.

Compile-time literal arrays. are defined by `#( )`, surrounding space-separated literals. Everything within the parentheses must be a compile-time constant. For example, `#(27 (true) abc 1+2)` is a literal array of 6 elements: the integer 27, the compile-time array containing the object `true` (non-changeable Boolean), the symbol `#abc`, the integer 1, the symbol `+` and the integer 2. Note that this is the same as `#(27 #(true) #abc 1 #+ 2)`.

Run-time dynamic arrays. Curly braces `{ }` define a dynamic array whose elements are expressions, separated by periods, and evaluated at run-time. So `{ 1. 2. 1 + 2 }` defines an array with elements 1, 2, and 3 the result of evaluating `1+2`.

Comments. are enclosed in double quotes " ". "hello" is a comment, not a string, and is ignored by the Pharo compiler. Comments may span multiple lines but they cannot be nested.

Local variable definitions. Vertical bars `| |` enclose the declaration of one or more local variables before the beginning of a method or a block body.

Assignment. `:=` assigns an object to a variable.

Blocks. Square brackets `[ ]` define a block, also known as a block closure or a lexical closure, which is a first-class object representing a function. As we shall see, blocks may take arguments (`[:i | ...]`) and can have local variables (`[| x | ...]`). Blocks also close over their definition environment, i.e., they can refer to variables that where reachable at the time of their definition.

Pragmas and primitives. `< primitive: ... >` is a method annotation. This specific one denotes the invocation of a virtual machine (VM) primitive. In the case of a primitive the code following it, it either to explain what the primitive is doing (for essential primitives) or is executed only if the primitive fails (for optional primitive). The same syntax of a message within `< >` is also used for other kinds of method annotations also called pragmas.

Unary messages. These consist of a single word (like `factorial`) sent to a receiver (like `3`). In `3 factorial`, `3` is the receiver, and `factorial` is the message selector.

Binary messages. These are messages sent to a receiver with a single argument, and whose selector looks like mathematical operator (for example: `+`). In `3 + 4`, the receiver is `3`, the message selector is `+`, and the argument is `4`.

Keyword messages. Their selectors consist of one or more keywords (like `raisedTo: modulo:`), each ending with a colon and taking a single argument. In the expression `2 raisedTo: 6 modulo: 10`, the message selector `raisedTo:modulo:` takes the two arguments `6` and `10`, one following each colon. We send the message to the receiver `2`.

Sequences of statements. A period or full-stop (`.`) is the statement separator. Putting a period between two expressions turns them into independent statements like in `x := 2. x := x + x`. Here we first assign value `2` to the variable `x`, and then duplicate its value by assigning a value of `x + x` to it.

Cascades. Semicolons (`;`) are used to send a cascade of messages to a single receiver. In `stream nextPutAll: 'Hello World'; close` we first send the keyword message `nextPutAll: 'Hello World'` to the receiver `stream`, and then we send the unary message `close` to the same receiver.

Method return. `^` is used to *return* a value from a method.

The basic classes `Number`, `Character`, `String` and `Boolean` are described in Chapter : Basic Classes.

For the purists.

We named the language elements expressions. At the level of the compiler it is not fully correct. For example returns are not expressions but statements. This is a detail when learning the language and more a concern of the language compiler and designer.

7.2 Pseudo-variables

In Pharo, there are 6 pseudo-variables: `nil`, `true`, `false`, `self`, `super`, and `thisContext`. They are called pseudo-variables because they are predefined and cannot be assigned to. `true`, `false`, and `nil` are constants, while the values of `self`, `super`, and `thisContext` vary dynamically as code is executed.

- `true` and `false` are the unique instances of classes `True` and `False` which are the subclasses of class `Boolean`. See Chapter : Basic Classes for more details.
- `self` always refers to the receiver of the message and denotes the object in which the corresponding method will be executed. Therefore, the value of `self` dynamically changes during the program execution, but can not be assigned in the code.
- `super` also refers to the receiver of the message too, but when you send a message to `super`, the method-lookup changes so that it starts from the superclass of the class containing the method that sends message to `super`. For further details see Chapter : The Pharo Object Model.
- `nil` is the undefined object. It is the unique instance of the class `UndefinedObject`. Instance variables, class variables and local variables are, by default, initialized to `nil`.
- `thisContext` is a pseudo-variable that represents the top frame of the execution stack and gives access to the current execution point. `thisContext` is normally not of interest to most programmers, but it is essential for implementing development tools such as the debugger, and it is also used to implement exception handling and continuations.

7.3 Messages and message sends

As we described, there are three kinds of messages in Pharo with predefined precedence. This distinction has been made to reduce the number of mandatory parentheses.

Here we give a brief overview on message kinds and ways for sending and executing them, while more detailed description is provided in Chapter : Understanding messages.

1. *Unary* messages take no argument. `1 factorial` sends the message `factorial` to the object 1. Unary message selectors consist of alphanumeric characters, and start with a lower case letter.
2. *Binary* messages take exactly one argument. `1 + 2` sends the message `+` with argument 2 to the object 1. Binary message selectors consist of one or more characters from the following set:

```
+ - / * ~ < > = @ % | & ? ,
```

1. *Keyword* messages take an arbitrary number of arguments. `2 raisedTo: 6 modulo: 10` sends the message consisting of the message selector `raisedTo:modulo:` and the arguments 6 and 10 to the object 2. Keyword message selectors consist of a series of alphanumeric keywords, where each keyword starts with a lower-case letter and ends with a colon.

Message precedence.

Unary messages have the highest precedence, then binary messages, and finally keyword messages, while brackets can be used to change the evaluation order.

Thus, in the following example we first send `factorial` to 3 which will give us result 6. Afterwards we send `+ 6` to 1 which gives the result 7, and finally we send `raisedTo: 7` to 2.

```
2 raisedTo: 1 + 3 factorial
>>> 128
```

Precedence aside, for the messages of the same kind, execution is strictly from left to right. Hence, as we have two binary messages, the following example return 9 and not 7.

```
1 + 2 * 3
>>> 9
```

Parentheses must be used to alter the order of evaluation as follows:

```
1 + (2 * 3)
>>> 7
```

7.4 Sequences and cascades

All expressions may be composed in sequences separated by period, while message sends may be also composed in cascades by semi-colons. A period separated sequence of expressions causes each expression in the series to be evaluated as a separate *statement*, one after the other.

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr
```

This will send `cr` to the `Transcript` object, then send to Transcript the message `show: 'hello world'`, and finally send it another `cr`, again.

When a series of messages is being sent to the *same* receiver, then this can be expressed more succinctly as a *cascade*. The receiver is specified just once, and the sequence of messages is separated by semi-colons as follows:

```
Transcript
  cr;
  show: 'hello world';
  cr
```

This cascade has precisely the same effect as the sequence in the previous example.

7.5 Method syntax

Whereas expressions may be evaluated anywhere in Pharo (for example, in a playground, in a debugger, or in a browser), methods are normally defined in a browser window, or in the debugger. Methods can also be filed in from an external medium, but this is not the usual way to program in Pharo.

Programs are developed one method at a time, in the context of a given class. A class is defined by sending a message to an existing class, asking it to create a subclass, so there is no special syntax required for defining classes.

Here is the method `lineCount` defined in the class `String`. The usual *convention* is to refer to methods as `ClassName>>methodName`. Here the method is then `String>>lineCount`. Note that `ClassName>>methodName` is not part of the Pharo syntax just a convention used in books to clearly define a method within a class in which it is defined.

```
String >> lineCount
  "Answer the number of lines represented by the receiver, where
    every cr adds one line."

  | cr count |
  cr := Character cr.
  count := 1 min: self size.
  self do: [:c | c == cr ifTrue: [count := count + 1]].
  ^ count
```

Syntactically, a method consists of:

1. the method pattern, containing the name (i.e., `lineCount`) and any parameters (none in this example)
2. comments which may occur anywhere, but the convention is to put one at the top that explains what the method does
3. declarations of local variables (i.e., `cr` and `count`); and
4. any number of expressions separated by dots (here there are four)

The execution of any expression preceded by a ^ (a caret or upper arrow, which is `Shift-6` for most keyboards) will cause the method to exit at that point, returning the value of the expression that follows the ^. A method

that terminates without explicitly returning value of some expression will implicitly return `self` object.

Parameters and local variables should always start with lower case letters. Names starting with upper-case letters are assumed to be global variables. Class names, like `Character`, for example, are simply global variables referring to the object representing that class.

7.6 Block syntax

Blocks (lexical closures) provide a mechanism to defer the execution of expressions. A block is essentially an anonymous function with a definition context. A block is executed by sending it the message `value`. The block answers the value of the last expression in its body, unless there is an explicit return (with ^) in which case it returns the value of the returned expression.

```
[ 1 + 2 ] value
>>> 3
```

```
[ 3 = 3 ifTrue: [ ^ 33 ]. 44 ] value
>>> 33
```

Blocks may have parameters each of which is declared with a leading colon. A vertical bar separates the parameters declaration from the body of the block. To evaluate a block with one parameter, you must send it the message `value:` with one argument. A two-parameter block must be evaluated by sending `value:value:` with two arguments, and so on, up to 4 arguments.

```
[ :x | 1 + x ] value: 2
>>> 3
```

```
[ :x :y | x + y ] value: 1 value: 2
>>> 3
```

If you have a block with more than four parameters, you must use `valueWithArguments:` and pass the arguments in an array. However, a block with a large number of parameters is often a sign of a design problem.

In blocks there may be also declared local variables, surrounded by vertical bars, just like local variable declarations in a method. Local variables are declared after arguments and vertical bar separator, and before the block body. In the following example, `x` `y` are parameters, and `z` is local variable.

```
[ :x :y |
  | z |
  z := x + y.
  z ] value: 1 value: 2
>>> 3
```

Blocks are actually lexical closures, since they can refer to variables of the surrounding environment. The following block refers to the variable `x` of its enclosing environment:

```
| x |
x := 1.
[ :y | x + y ] value: 2
>>> 3
```

Blocks are instances of the class `BlockClosure`. This means that they are objects, so they can be assigned to variables and passed as arguments just like any other object.

7.7 Conditionals and loops

Pharo offers no special syntax for control constructs. Instead, these are typically expressed by sending messages to booleans, numbers and collections, with blocks as arguments.

Some conditionals

Conditionals are expressed by sending one of the messages `ifTrue:`, `ifFalse:` or `ifTrue:ifFalse:` to the result of a boolean expression. See Chapter : Basic Classes, for more about booleans.

```
(17 * 13 > 220)
  ifTrue: [ 'bigger' ]
  ifFalse: [ 'smaller' ]
>>>'bigger'
```

Some loops

Loops are typically expressed by sending messages to blocks, integers or collections. Since the exit condition for a loop may be repeatedly evaluated, it should be a block rather than a boolean value. Here is an example of a very procedural loop:

```
n := 1.
[ n < 1000 ] whileTrue: [ n := n*2 ].
n
>>> 1024
```

`whileFalse:` reverses the exit condition.

```
n := 1.
[ n > 1000 ] whileFalse: [ n := n*2 ].
n
>>> 1024
```

`timesRepeat:` offers a simple way to implement a fixed number of iterations through the loop body:

```
n := 1.
10 timesRepeat: [ n := n*2 ].
n
>>> 1024
```

We can also send the message `to:do:` to a number which then acts as the initial value of a loop counter. The two arguments are the upper bound, and a block that takes the current value of the loop counter as its argument:

```
result := String new.
1 to: 10 do: [:n | result := result, n printString, ' '].
result
>>> '1 2 3 4 5 6 7 8 9 10 '
```

High-order iterators

Collections comprise a large number of different classes, many of which support the same protocol. The most important messages for iterating over collections include `do:`, `collect:`, `select:`, `reject:`, `detect:` and `inject:into:`. These messages represent high-level iterators that allow one to write very compact code.

An **Interval** is a collection that lets one iterate over a sequence of numbers from the starting point to the end. `1 to: 10` represents the interval from 1 to 10. Since it is a collection, we can send the message `do:` to it. The argument is a block that is evaluated for each element of the collection.

```
result := String new.
(1 to: 10) do: [:n | result := result, n printString, ' '].
result
>>> '1 2 3 4 5 6 7 8 9 10 '
```

`collect:` builds a new collection of the same size, transforming each element. You can think of `collect:` as the Map in the MapReduce programming mode).

```
(1 to:10) collect: [ :each | each * each ]
>>> #(1 4 9 16 25 36 49 64 81 100)
```

`select:` and `reject:` build new collections, each containing a subset of the elements of the iterated collection that satisfies, or not, respectively, the boolean block condition.

`detect:` returns the first element in the collection that satisfies the condition.

Don't forget that strings are also collections (of characters), so you can iterate over all the characters.

```
'hello there' select: [ :char | char isVowel ]
>>> 'eoee'
```

```
'hello there' reject: [ :char | char isVowel ]
>>> 'hll thr'
```

```
'hello there' detect: [ :char | char isVowel ]
>>> $e
```

Finally, you should be aware that collections also support a functional-style fold operator in the `inject:into:` method. You can also think of it as the Reduce in the MapReduce programming model. This lets you generate a cumulative result using an expression that starts with a seed value and injects each element of the collection. Sums and products are typical examples.

```
(1 to: 10) inject: 0 into: [ :sum :each | sum + each]
>>> 55
```

This is equivalent to `0+1+2+3+4+5+6+7+8+9+10`.

More about collections can be found in Chapter : Collections.

7.8 Method annotations: Primitives and pragmas

In Pharo methods can be annotated too. Method annotation are delimitated by < and >. There are used for two main scenarios: execution specific metadata for the primitives of the language and metadata.

Primitives

In Pharo everything is an object, and everything happens by sending messages. Nevertheless, at certain points we hit rock bottom. Certain objects can only get work done by invoking virtual machine primitives. Such primitives are essential primitives since they cannot be expressed in Pharo.

For example, the following are all implemented as primitives: memory allocation (`new`, `new:`), bit manipulation (`bitAnd:`, `bitOr:`, `bitShift:`), pointer and integer arithmetic (+, -, <, >, *, /, =, ==...), and array access (`at:`, `at:put:`).

When a method with a primitive is executed, the primitive code is executed in place of the method. A method using such a primitive may include additional Pharo code, which will be executed only if the primitive fails (for the case the primitive is an optional one).

In the following example, we see the code for `SmallInteger>>+`. If the primitive fails, the expression `super + aNumber` will be evaluated and its value returned.

```
+ aNumber
  "Primitive. Add the receiver to the argument and answer with the
    result
  if it is a SmallInteger. Fail if the argument or the result is not
    a
  SmallInteger Essential No Lookup. See Object documentation
    whatIsAPrimitive."

  <primitive: 1>
  ^ super + aNumber
```

Pragmas.

In Pharo, the angle bracket syntax is also used for method annotations called pragmas. Once a method has been annotated with a pragma, the annotations can be collected using a collection (see the class `PragmaCollector`).

7.9 Chapter summary

- Pharo has only six reserved identifiers known as pseudo-variables: `true`, `false`, `nil`, `self`, `super`, and `thisContext`.
- There are five kinds of literal objects: numbers (5, 2.5, 1.9e15, 2r111), characters (`$a`), strings (`'hello'`), symbols (`#hello`), and arrays (`#('hello' #hi)` or `{ 1 . 2 . 1 + 2 }`)
- Strings are delimited by single quotes, comments by double quotes. To get a quote inside a string, double it.
- Unlike strings, symbols are guaranteed to be globally unique.
- Use `#( ... )` to define a literal array at compile time. Use `{ ... }` to define a dynamic array at runtime. Note that `#(1+2) size >>> 3`, but `{12+3} size >>> 1`. To observe why, compare `#(12+3) inspect` and `{1+2} inspect`.
- There are three kinds of messages: unary (e.g., `1 asString`, `Array new`), binary (e.g., `3 + 4`, `'hi', ' there'`), and keyword (e.g., `'hi' at: 2 put: $o`)
- A cascaded message send is a sequence of messages sent to the same target, separated by semi-colons: `OrderedCollection new add: #calvin; add: #hobbes; size >>> 2`
- Local variables declarations are delimited by vertical bars. Use `:=` for assignment. `|x| x := 1`
- Expressions consist of message sends, cascades and assignments, evaluated left to right (and optionally grouped with parentheses). Statements are expressions separated by periods.

- Block closures are expressions enclosed in square brackets. Blocks may take arguments and can contain temporary variables. The expressions in the block are not evaluated until you send the block a value message with the correct number of arguments. `[ :x | x + 2 ] value: 4`
- There is no dedicated syntax for control constructs, just messages whose sends conditionally evaluate blocks.

CHAPTER 8

Understanding message syntax

Although Pharo's message syntax is extremely simple, it is unconventional and can take some time getting used to. This chapter offers some guidance to help you get acclimatized to the syntax for sending messages. If you already feel comfortable with the syntax, you may choose to skip this chapter, or come back to it later. The Pharo's syntax is closed to the one of Smalltalk, so Smalltalk programmers can be familiar with Pharo's syntax.

8.1 Identifying messages

In Pharo, except for the syntactic elements listed in Chapter 7 (:= ^ . ; # () {} [: |]), everything is a message send. You can define operators like + for your own classes, but all operators, existing and defined ones, have the same precedence. In fact, in Pharo there is no operators! Just messages of a given kind: *unary*, *binary* or *keywords*. Moreover, you cannot change the arity of a message selector. The selector "-" is always the selector of a binary message; there is no way to have a unary - for unary messages.

In Pharo, the order in which messages are sent is determined by the *kind* of message. There are just three kinds of messages: *unary*, *binary*, and *keyword messages*. Unary messages are always sent first, then binary messages and finally keyword ones. As in most languages, parentheses are used to change the execution order. These rules make Pharo code as easy to read as possible. And most of the time you do not have to think about the rules.

As most computation in Pharo is done by message passing, correctly identifying messages is crucial. The following terminology will help us:

- A message is composed of a message *selector* and optional message arguments.

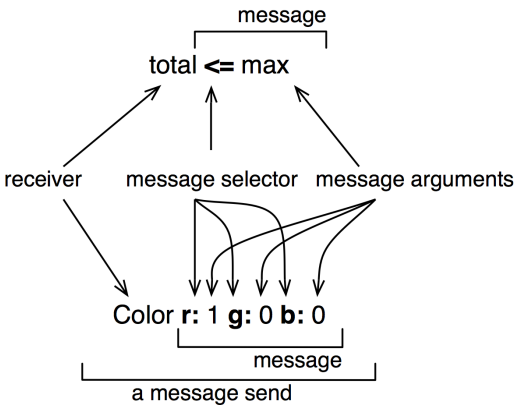

Figure 8-1 Two message sends composed of a receiver, a method selector, and a set of arguments.

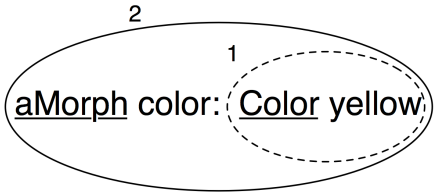

Figure 8-2 Two messages: `Color yellow` and `aMorph color: Color yellow`.

- A message is sent to a *receiver*.
- The combination of a message and its receiver is called a *message send* as shown in Figure 8-1.

A message is always sent to a receiver, which can be a single literal, a block or a variable or the result of evaluating another message. To help you identify the receiver of a message, we will underline it for you. We will also surround each message send with an ellipse and number message sends starting from the first one that will be sent to help you see the order in which messages are sent.

Figure 8-2 represents two message sends, `Color yellow` and `aMorph color: Color yellow`, hence there are two ellipses. The message send `Color yellow` is executed first so its ellipse is numbered `1`. There are two receivers: `aMorph` which receives the message `color: ...` and `Color` which receives the message `yellow`. Both receivers are underlined.

A receiver can be the first element of a message, such as `100` in the message send `100 + 200` or `Color` in the message send `Color yellow`. However, a

receiver can also be the result of other messages. For example in the message `Pen new go: 100`, the receiver of the message `go: 100` is the object returned by the message send `Pen new`. In all the cases, a message is sent to an object called the *receiver* which may be the result of another message send.

Message send	Message type	Result
`Color yellow`	unary	Creates color yellow.
`aPen go: 100`	keyword	The pen moves forward 100 pixels.
`100 + 20`	binary	100 is increased by 20
`Browser open`	unary	Opens a new browser.
`Pen new go: 100`	unary and keyword	Creates and moves a pen 100 pixels forward.
`aPen go: 100 + 20`	keyword and binary	The pen moves forward 120 pixels.

The table shows several examples of message sends. You should note that:

- Not all message sends have arguments. Unary messages like `open` do not have arguments.
- Single keyword and binary messages like `go: 100` and `+ 20` each have one argument.
- There are also simple messages and composed ones. `Color yellow` and `100 + 20` are simple: a message is sent to an object, while the message send `aPen go: 100 + 20` is composed of two messages: `+ 20` is sent to `100` and `go:` is sent to `aPen` with the argument being the result of the first message.
- A receiver can be an expression (such as an assignment, a message send or a literal) which returns an object. In `Pen new go: 100`, the message `go: 100` is sent to the object that results from the execution of the message send `Pen new`.

8.2 Three kinds of messages

Pharo defines a few simple rules to determine the order in which the messages are sent. These rules are based on the distinction between 3 different kinds of messages:

- *Unary messages* are messages that are sent to an object without any other information. For example in `3 factorial`, `factorial` is a unary message. A sent unary message may execute a basic unary operation or an arbitrary functionality, but it is always sent without arguments.
- *Binary messages* are messages consisting of operators (often arithmetic) and executing basic binary operations.

They are binary because they always involve only two objects: the receiver and the argument object. For example in `10 + 20`, `+` is a binary message sent to the receiver `10` with argument `20`.

- *Keyword messages* are messages consisting of one or more keywords, each ending with a colon (`:`) and taking an argument. For example in `anArray at: 1 put: 10`, the keyword `at:` takes the argument `1` and the keyword `put:` takes the argument `10`.

It is important to note that:

- There are no keyword messages that are sent without arguments. All messages that are sent without arguments are unary ones.
- There is a difference between keyword messages that are sent with exactly one argument and binary ones - a keyword message send may execute an arbitrary functionality.

Unary messages

Unary messages are messages that do not require any argument. They follow the syntactic template: `receiver messageName`. The selector is simply made up of a succession of characters not containing `:` (*e.g.*, `factorial`, `open`, `class`).

```
89 sin
>>> 0.860069405812453
```

```
3 sqrt
>>> 1.732050807568877
```

```
Float pi
>>> 3.141592653589793
```

```
'blop' size
>>> 4
```

```
true not
>>> false
```

```
Object class
>>> Object class  "The class of Object is Object class (BANG)"
```

Important Unary messages follow the syntactic template: receiver **selector**

Binary messages

Binary messages are messages that require exactly one argument *and* whose selector consists of a sequence of one or more characters from the set: `+`, `-`, `*`, `/`, `&`, `=`, `>`, `|`, `<`, `~`, and `@`. Note that `--` is not allowed for parsing reasons.

```
100@100
>>> 100@100  "creates a Point object"
```

```
3 + 4
>>> 7
```

```
10 - 1
>>> 9
```

```
4 <= 3
>>> false
```

```
(4/3) * 3 == 4
>>> true  "equality is just a binary message, and Fractions are
    exact"
```

```
(3/4) == (3/4)
>>> false  "two equal Fractions are not the same object"
```

Important Binary messages follow the syntactic template: receiver **selector** argument

Keyword messages

Keyword messages are messages that require one or more arguments and whose selector consists of one or more keywords each ending in `:`.

In the following example, message `between:and:` is composed of two keywords: `between:` and `and:`. The full message is `between:and:` and it is sent to number.

```
2 between: 0 and: 10
>>> true
```

Each keyword takes an argument. Hence `r:g:b:` is a message with three arguments, `playFileNamed:` and `at:` are messages with one argument, and `at:put:` is a message with two arguments. To create an instance of the class `Color` one can use the message `r:g:b:` as in `Color r: 1 g: 0 b: 0`, which creates the color red. Note that the colons are part of the selector.

```
Color r: 1 g: 0 b: 0
>>> Color red  "creates a new color"
```

In a Java like syntax, the Pharo message send `Color r: 1 g: 0 b: 0` would correspond to a method invocation written as `Color.rgb(1,0,0)`.

```
1 to: 10
>>> (1 to: 10)  "creates an interval"
```

```
| nums |
nums := Array newFrom: (1 to: 5).
nums at: 1 put: 6.
nums
```

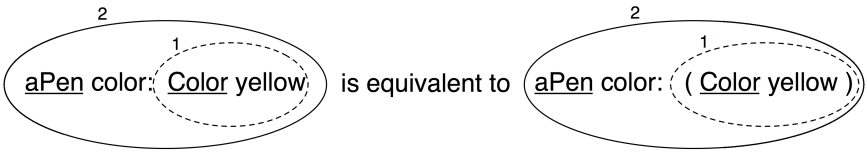

Figure 8-3 Unary messages are sent first so Color yellow is sent. This returns a color object which is passed as argument of the message aPen color:.

```
>>> #(6 2 3 4 5)
```

> **Important** Keyword messages follow the syntactic template: receiver **selectorWordOne:** argumentOne **wordTwo:** argumentTwo ... **wordN:** argumentN

8.3 Message composition

The three kinds of messages each have different precedence, which allows them to be composed in an elegant way.

- Unary messages are always sent first, then binary messages and finally keyword messages.
- Messages in parentheses are sent prior to any kind of messages.
- Messages of the same kind are evaluated from left to right.

These rules lead to a very natural reading order. Now if you want to be sure that your messages are sent in the order that you want you can always put more parentheses as shown in Figure 8-3. In this figure, the message `yellow` is an unary message and the message `color:` a keyword message, therefore the message send `Color yellow` is sent first. However as message sends in parentheses are sent first, putting (unnecessary) parentheses around `Color yellow` just emphasizes that it will be sent first. The rest of the section illustrates each of these points.

Unary > Binary > Keywords

Unary messages are sent first, then binary messages, and finally keyword messages. We also say that unary messages have a higher priority over the other kinds of messages.

> **Important** `Unary > Binary > Keyword`

As these examples show, Pharo's syntax rules generally ensure that message sends can be read in a natural way:

```
1000 factorial / 999 factorial
>>> 1000
```

```
2 raisedTo: 1 + 3 factorial
>>> 128
```

Unfortunately the rules are a bit too simplistic for arithmetic message sends, so you need to introduce parentheses whenever you want to impose a priority over binary operators:

```
1 + 2 * 3
>>> 9
```

```
1 + (2 * 3)
>>> 7
```

We will dedicate a section to arithmetic inconsistencies.

The following example, which is a bit more complex (!), offers a nice illustration that even complicated expressions can be read in a natural way:

```
[:aClass | aClass methodDict keys select: [:aMethod |
    (aClass>>aMethod) isAbstract ]] value: Boolean
>>> an IdentitySet(#or: #| #and: #& #ifTrue: #ifTrue:ifFalse:
    #ifFalse: #not #ifFalse:ifTrue:)
```

Here we want to know which methods of the `Boolean` class are abstract. We ask some argument class, `aClass`, for the keys of its method dictionary, and select those methods of that class that are abstract. Then we bind the argument `aClass` to the concrete value `Boolean`. We need parentheses only to send the binary message >>, which selects a method from a class, before sending the unary message `isAbstract` to that method. The result shows us which methods must be implemented by `Boolean`'s concrete subclasses `True` and `False`.

In fact, we could also have written the equivalent but simpler expression: `Boolean methodDict select: [:each | each isAbstract] thenCollect: [:each | each selector]`.

Example. In the message `aPen color: Color yellow`, there is one *unary* message `yellow` sent to the class `Color` and a *keyword* message `color:` sent to `aPen`. Unary messages are sent first so the message send `Color yellow` is sent (1). This returns a color object which is passed as argument of the message `aPen color: aColor` (2). Figure 8-3 shows graphically how messages are sent.

```
Decomposing the execution of aPen color: Color yellow
         aPen color: Color yellow
(1)         Color yellow         "unary message is sent first"
               >>> aColor
(2)         aPen color: aColor   "keyword message is sent next"
```

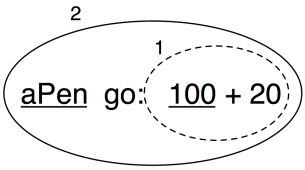

Figure 8-4 Binary messages are sent before keyword messages.

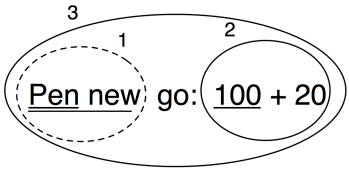

Figure 8-5 Decomposing Pen new go: 100 + 20.

Example. In the message `aPen go: 100 + 20`, there is a *binary* message `+ 20` and a *keyword* message `go:`. Binary messages are sent prior to keyword messages so `100 + 20` is sent first (1): the message `+ 20` is sent to the object `100` and returns the number `120`. Then the message `aPen go: 120` is sent with `120` as argument (2). The following example shows how the message send is executed.

```
        aPen go: 100 + 20
(1)        100 + 20          "binary message first"
              >>>   120
(2)        aPen go: 120      "then keyword message"
```

Example. As an exercise we let you decompose the execution of the message send `Pen new go: 100 + 20` which is composed of one unary, one keyword and one binary message (see Figure 8-5).

Parentheses first

Parenthesised messages are sent prior to other messages.

Important (Msg) > Unary > Binary > Keyword

Here are some examples.

The first example shows that parentheses are not needed when the order is the one we want, *i.e.* that result is the same if we write this with or without parentheses. Here we compute tangent of 1.5, then we round it and convert it as a string.

```
1.5 tan rounded asString = (((1.5 tan) rounded) asString)
>>> true
```

The second example shows that `factorial` is executed prior to the sum and if we want to first perform the sum of 3 and 4 we should use parentheses as shown below.

```
3 + 4 factorial
>>> 27    "(not 5040)"
```

```
(3 + 4) factorial
>>> 5040
```

Similarly in the following example, we need the parentheses to force sending `lowMajorScaleOn:` before `play`.

```
(FMSound lowMajorScaleOn: FMSound clarinet) play
"(1) send the message clarinet to the FMSound class to create a
    clarinet sound.
 (2) send this sound to FMSound as argument to the lowMajorScaleOn:
    keyword message.
 (3) play the resulting sound."
```

Example. The message send `(65@325 extent: 134@100) center` returns the center of a rectangle whose top left point is (65, 325) and whose size is 134*100. The following example shows how the message is decomposed and sent. First the message between parentheses is sent: it contains two binary messages `65@325` and `134@100` that are sent first and return points, and a keyword message `extent:` which is then sent and returns a rectangle. Finally the unary message `center` is sent to the rectangle and a point is returned. Evaluating the message without parentheses would lead to an error because the object `100` does not understand the message `center`.

```
Example of Parentheses.
      (65@325 extent: 134@100) center
(1)   65@325
    "binary"
    >>> aPoint
(2)                                134@100
    "binary"
                                 >>> anotherPoint
(3)   aPoint extent: anotherPoint                        "keyword"
      >>> aRectangle
(4)   aRectangle center                                      "unary"
      >>> 132@375
```

From left to right

Now we know how messages of different kinds or priorities are handled. The final question to be addressed is how messages with the same priority are

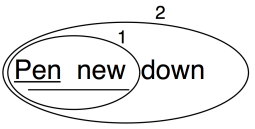

Figure 8-6 Decomposing Pen new down.

sent. They are sent from the left to the right. Note that you already saw this behaviour in the example `1.5 tan rounded asString` where all unary messages are sent from left to right which is equivalent to `(((1.5 tan) rounded) asString)`.

> **Important** When the messages are of the same kind, the order of execution is from left to right.

Example. In the message sends `Pen new down` all messages are unary messages, so the leftmost one, `Pen new`, is sent first. This returns a newly created pen to which the second message `down` is sent, as shown in Figure 8-6.

Arithmetic inconsistencies

The message composition rules are simple. There is no notion of mathematical precedence because arithmetic messages are just messages as any other ones. So their result may look in inconsistent when executed them. Here we see the common situations where extra parentheses are needed.

```
3 + 4 * 5
>>> 35    "(not 23)  Binary messages sent from left to right"
```

```
3 + (4 * 5)
>>> 23
```

```
1 + 1/3
>>> (2/3)    "and not 4/3"
```

```
1 + (1/3)
>>> (4/3)
```

```
1/3 + 2/3
>>> (7/9)    "and not 1"
```

```
(1/3) + (2/3)
>>> 1
```

Example. In the message sends `20 + 2 * 5`, there are only binary messages `+` and `*`. However in Pharo there is no specific priority for the operations `+` and `*`. They are just binary messages, hence `*` does not have priority

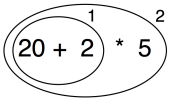

Figure 8-7 Default execution order.

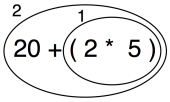

Figure 8-8 Changing default execution order using parentheses.

over +. Here the leftmost message + is sent first (1) and then the * is sent to the result as shown in below.

```
"As there is no priority among binary messages, the leftmost message
    + is evaluated first even if by the rules of arithmetic the *
    should be sent first."

     20 + 2 * 5
(1)  20 + 2 >>> 22
(2)  22     * 5 >>> 110
```

As shown in the previous example the result of this message send is not `30` but `110`. This result is perhaps unexpected but follows directly from the rules used to send messages. This is the price to pay for the simplicity of the model. To get the correct result, we should use parentheses. When messages are enclosed in parentheses, they are evaluated first. Hence the message send `20 + (2 * 5)` returns the result as shown.

```
Decomposing 20 + (2 * 5)
"The messages surrounded by parentheses are evaluated first
    therefore * is sent prior
to + which produces the correct behaviour."

    20 + (2 * 5)
(1)      (2 * 5) >>> 10
(2) 20 + 10      >>> 30
```

Important In Pharo, arithmetic operators such as + and * do not have different priority. + and * are just binary messages, therefore * does not have priority over +. Use parentheses to obtain the desired result.

To do is this a "caption" of the table? Figures around move and separate it from the table...

Table : Message sends and their fully parenthesized equivalents

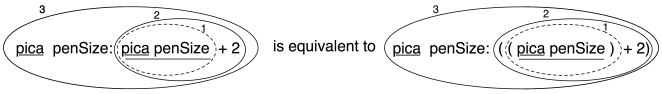

Figure 8-9 Equivalent messages using parentheses.

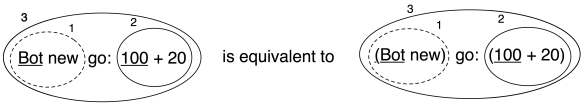

Figure 8-10 Equivalent messages using parentheses.

Implicit precedence	Explicitly parenthesized equivalent
aPen color: Color yellow	aPen color: (Color yellow)
aPen go: 100 + 20=	Pen go: (100 + 20)
aPen penSize: aPen penSize + 2	aPen penSize: ((aPen penSize) + 2)
2 factorial + 4	(2 factorial) + 4

Note that the first rule stating that unary messages are sent prior to binary and keyword messages avoids the need to put explicit parentheses around them. Table above shows message sends written following the rules and equivalent message sends if the rules would not exist. Both message sends result in the same effect or return the same value.

8.4 Hints for identifying keyword messages

Often beginners have problems understanding when they need to add parentheses. Let's see how keywords messages are recognized by the compiler.

Parentheses or not?

The characters [,], (and) delimit distinct areas. Within such an area, a keyword message is the longest sequence of words terminated by `:` that is not cut by the characters `.`, or `;`. When the characters `[` and `]`, `(` and `)` surround some words with colons, these words participate in the keyword message *local* to the area defined.

In this example, there are two distinct keyword messages: `rotatedBy:magnify:smoothing:` and `at:put:`.

```
aDict
   at: (rotatingForm
          rotateBy: angle
          magnify: 2
          smoothing: 1)
```

```
   put: 3
```

Hints. If you have problems with these precedence rules, you may start simply by putting parentheses whenever you want to distinguish two messages having the same precedence.

The following piece of code does not require parentheses because the message `isNil` is unary hence it is sent prior to the keyword message `ifTrue:`.

```
(x isNil)
   ifTrue:[...]
```

The following piece of code requires parentheses because the messages `includes:` and `ifTrue:` are both keyword messages.

```
ord := OrderedCollection new.
(ord includes: $a)
   ifTrue:[...]
```

Without parentheses the unknown message `includes:ifTrue:` would be sent to the collection `ord`!

When to use [ ] or ( )

You may also have problems understanding when to use square brackets rather than parentheses. The basic principle is that you should use `[]` when you do not know how many times, potentially zero, an expression should be evaluated. `[expression]` will create block closure (*i.e.*, an object) from `expression`, which may be evaluated any number of times (possibly zero), depending on the context. Note that an expression can either be a message send, a variable, a literal, an assignment or a block.

Hence the conditional branches of `ifTrue:` or `ifTrue:ifFalse:` require blocks. Following the same principle both the receiver and the argument of a `whileTrue:` message require the use of square brackets since we do not know how many times either the receiver or the argument should be evaluated.

Parentheses, on the other hand, only affect the order of sending messages. So in `(expression)`, the `expression` will *always* be evaluated exactly once.

```
[ x isReady ] whileTrue: [ y doSomething ]  "both the receiver and
    the argument must be blocks"
4 timesRepeat: [ Beeper beep ]              "the argument is
    evaluated more than once, so must be a block"
(x isReady) ifTrue: [ y doSomething ]       "receiver is evaluated
    once, so is not a block
                                            argument does not have
    to be evaluated not even once,
                      so it is a block"
```

8.5 Expression sequences

Expressions (*i.e.*, message sends, assignments, ...) separated by periods are evaluated in sequence. Note that there is no period between a variable declaration and the following expression. The value of a sequence is the value gained by the evaluation of the last expression in the sequence. The values returned by all the expressions except the last one are ignored at the end. Note that the period is a separator and not a terminator. Therefore, a final period is optional.

```
| box |
box := 20@30 corner: 60@90.
box containsPoint: 40@50
>>> true
```

8.6 Cascaded messages

Pharo offers a way to send multiple messages to the same receiver, without stating it multiple times, by using a semicolon separator (`;`). This is called the cascade in Pharo jargon.

Syntactically a cascade is represented as follows:

```
aReceiverExpression msg1 ; msg2 ; msg3
```

Examples. You can program in Pharo without using cascades. It just forces you to repeat the receiver of the message. The following code snippets are equivalent:

```
Transcript show: 'Pharo is '.
Transcript show: 'fun '.
Transcript cr.
```

```
Transcript
   show: 'Pharo is';
   show: 'fun ';
   cr
```

In fact the receiver of all the cascaded messages is the receiver of the first message involved in a cascade. Note that the object receiving the cascaded messages can itself be the result of a message send. In the following example, the first cascaded message is `setX:setY` since it is followed by a cascade. The receiver of the cascaded message `setX:setY:` is the newly created point resulting from the execution of `Point new`, and *not* `Point`. The subsequent message `isZero` is sent to that same receiver.

```
Point new setX: 25 setY: 35; isZero
>>> false
```

8.7 Chapter summary

- A message is always sent to an object named the *receiver* which may be the result of other message sends.
- Unary messages are messages that do not require any argument. They are of the form: selector.
- Binary messages are messages that involve two objects, the receiver and another object, *and* whose selector is composed of one or more characters from the following list: `+`, `-`, `*`, `/`, `|`, `&`, `=`, `>`, `<`, `~`, and `@`. They are of the form: receiver **selector** argument
- Keyword messages are messages that involve more than one object and that contain at least one colon character (`:`). They are of the form: receiver **selectorKeywordOne:** argumentOne **KeywordTwo:** argumentTwo ... **KeywordN:** argumentN
- **Rule One.** Unary messages are sent first, then binary messages, and finally keyword messages.
- **Rule Two.** Messages in parentheses are sent before any others.
- **Rule Three.** When the messages are of the same kind, the order of execution is from left to right.
- In Pharo, traditional arithmetic operators such as `+` and * have the same priority. `+` and `*` are just binary messages, therefore `*` does not have priority over `+`. You must use parentheses to obtain a correct result.

CHAPTER 9

The Pharo object model

The Pharo language model is inspired by the one of Smalltalk. It is simple and uniform: everything is an object, and objects communicate only by sending messages to each other. Instance variables are private to the object and methods are all public and dynamically looked up (late-bound).

In this chapter, we present the core concepts of the Pharo object model. We sorted the sections of this chapter to make sure that the most important points appear first. We revisit concepts such as `self`, `super` and precisely define their semantics. Then we discuss the consequences of representing classes as objects. This will be extended in Chapter: Classes and Metaclasses.

9.1 The rules of the core model

The object model is based on a set of simple rules that are applied *uniformly* and systematically without any exception. The rules are as follows:

Rule 1 Everything is an object.

Rule 2 Every object is an instance of a class.

Rule 3 Every class has a superclass.

Rule 4 Everything happens by sending messages.

Rule 5 Method lookup follows the inheritance chain.

Rule 6 Classes are objects too and follow exactly the same rules.

Let us look at each of these rules in detail.

Listing 9-1 Sending `+ 4` to 3, yields the object 7.

```
3 + 4
>>> 7
```

Listing 9-2 Sending `factorial` to `20`, yields a large number.

```
20 factorial
>>> 2432902008176640000
```

9.2 Everything is an Object

The mantra *everything is an object* is highly contagious. After only a short while working with Pharo, you will become surprised how this rule simplifies everything you do. Integers, for example, are objects too, so you send messages to them, just as you do to any other object. At the end of this chapter, we added an implementation note on the object implementation for the curious reader.

Here are two examples.

The object `7` is different than the object returned by `20 factorial`. `7` is an instance of `SmallInteger` while `20 factorial` is an instance of `LargePositiveInteger`. But because they are both polymorphic objects (they know how to answer to the same set of messages), none of the code, not even the implementation of `factorial`, needs to know about this.

Coming back to *everything is an object* rule, perhaps the most fundamental consequence of this rule is that classes are objects too. Classes are not second-class objects: they are really first-class objects that you can send messages to, inspect, and change. This means that Pharo is a truly reflective system, which gives a great deal of expressive power to developers.

> **Important** Classes are objects too. We interact the same way with classes and objects, simply by sending messages to them.

9.3 Every object is an instance of a class

Every object has a class and you can find out which one by sending message `class` to it.

```
1 class
>>> SmallInteger
```

```
20 factorial class
>>> LargePositiveInteger
```

```
'hello' class
>>> ByteString
```

```
(4@5) class
>>> Point
```

```
Object new class
>>> Object
```

A class defines the *structure* of its instances via instance variables, and the *behavior* of its instances via methods. Each method has a name, called its *selector*, which is unique within the class.

Since *classes are objects*, and *every object is an instance of a class*, it follows that classes must also be instances of classes. A class whose instances are classes is called a *metaclass*. Whenever you create a class, the system automatically creates a metaclass for you. The metaclass defines the structure and behavior of the class that is its instance. You will not need to think about metaclasses 99% of the time, and may happily ignore them. We will have a closer look at metaclasses in Chapter : Classes and Metaclasses.

9.4 Instance structure and behavior

Now we will briefly present how we specify the structure and behavior of instances.

Instance variables

Instance variables in Pharo are private to the *instance* itself. This is in contrast to Java and C++, which let instance variables (also known as *fields* or *member variables*) to be accessed by any other instance that happens to be of the same class. We say that the *encapsulation boundary* of objects in Java and C++ is the class, whereas in Pharo it is the instance.

In Pharo, two instances of the same class cannot access each other's instance variables unless the class defines *accessor methods*. There is no language syntax that provides direct access to the instance variables of any other object. (Actually, a mechanism called reflection does provide a way to ask another object for the values of its instance variables; meta-programming is intended for writing tools like the object inspector, whose sole purpose is to look inside other objects.)

Instance variables can be accessed by name in any of the instance methods of the class that defines them, and also in the methods defined in its subclasses. This means that Pharo instance variables are similar to *protected* variables in C++ and Java. However, we prefer to say that they are private, because it is considered bad style in Pharo to access an instance variable directly from a subclass.

Listing 9-3 Distance between two points.

```
Point >> distanceTo: aPoint
    "Answer the distance between aPoint and the receiver."

    | dx dy |
    dx := aPoint x - x.
    dy := aPoint y - y.
    ^ ((dx * dx) + (dy * dy)) sqrt
```

Instance encapsulation example

The method `distanceTo:` of the class `Point` computes the distance between the receiver and another point. The instance variables `x` and `y` of the receiver are accessed directly by the method body. However, the instance variables of the other point must be accessed by sending it the messages `x` and `y`.

```
1@1 dist: 4@5
>>> 5.0
```

The key reason to prefer instance-based encapsulation to class-based encapsulation is that it enables different implementations of the same abstraction to coexist. For example, the method `distanceTo:` doesn't need to know or care whether the argument `aPoint` is an instance of the same class as the receiver. The argument object might be represented in polar coordinates, or as a record in a database, or on another computer in a distributed system. As long as it can respond to the messages `x` and `y`, the code of method `distanceTo:` (shown above) will still work.

Methods

All methods are *public* and *virtual* (i.e., dynamically looked up). There is no static methods in Pharo. Methods can access all instance variables of the object. Some developers prefer to access instance variables only through accessors. This practice has some value, but it also clutters the interface of your classes, and worse, exposes its private state to the world.

To ease class browsing, methods are grouped into *protocols* that indicate their intent. Protocols have no semantics from the language view point. They are just folders in which methods are stored. Some common protocol names have been established by convention, for example, `accessing` for all accessor methods and `initialization` for establishing a consistent initial state for the object. The protocol `private` is sometimes used to group methods that should not be seen from outside. Nothing, however, prevents you from sending a message that is implemented by such a "private" method.

Listing 9-4 The definition of the class `Point`.

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Kernel-BasicObjects'
```

9.5 Every class has a superclass

Each class in Pharo inherits its behaviour and the description of its structure from a single *superclass*. This means that Pharo offers single inheritance.

Here are some examples showing how can we navigate the hierarchy.

```
SmallInteger superclass
>>> Integer
```

```
Integer superclass
>>> Number
```

```
Number superclass
>>> Magnitude
```

```
Magnitude superclass
>>> Object
```

```
Object superclass
>>> ProtoObject
```

```
ProtoObject superclass
>>> nil
```

Traditionally the root of an inheritance hierarchy is the class `Object` (since everything is an object). In Pharo, the root is actually a class called `ProtoObject`, but you will normally not pay any attention to this class. `ProtoObject` encapsulates the minimal set of messages that all objects *must* have and `ProtoObject` is designed to raise as many as possible errors (to support proxy definition). However, most classes inherit from `Object`, which defines many additional messages that almost all objects understand and respond to. Unless you have a very good reason to do otherwise, when creating application classes you should normally subclass `Object`, or one of its subclasses.

A new class is normally created by sending the message `subclass: instanceVariableNames: ...` to an existing class as shown in 9-4. There are a few other methods to create classes. To see what they are, have a look at `Class` and its `subclass creation` protocol.

9.6 Everything happens by sending messages

This rule captures the essence of programming in Pharo.

Listing 9-5 Sending message + with argument 4 to integer 3.

```
3 + 4
>>> 7
```

Listing 9-6 Sending message + with argument 4 to point (1@2).

```
(1@2) + 4
>>> 5@6
```

In procedural programming (and in some static features of some object-oriented languages such as Java), the choice of which piece of code to execute when a procedure is called is made by the caller. The caller chooses the procedure to execute *statically*, by name. In such a case there is no lookup or dynamicity involved.

In Pharo, we do *not* "invoke methods". Instead, we *send messages.* This is just a terminology point but it is significant. It implies that this is not the responsibility of the client to select the method to be executed, it is the one of the receiver of the message.

When sending a message, we do not decide which method will be executed. Instead, we *tell* an object to do something for us by sending it a message. A message is nothing but a name and a list of arguments. The receiver then decides how to respond by selecting its own *method* for doing what was asked. Since different objects may have different methods for responding to the same message, the method must be chosen *dynamically*, when the message is received.

As a consequence, we can send the *same message* to different objects, each of which may have *its own method* for responding to the message. We do not tell the `SmallInteger 3` or the `Point (1@2)` how to respond to the message `+ 4`. Each has its own method for `+`, and responds to `+ 4` accordingly.

About other computation.

Nearly everything in Pharo happens by sending messages. At some point action must take place:

- *Variable declarations* are not message sends. In fact, variable declarations are not even executable. Declaring a variable just causes space to be allocated for an object reference.
- *Variable accesses* are just access to the value of a variable.
- *Assignments* are not message sends. An assignment to a variable causes that variable name to be freshly bound in the scope of its definition.
- *Returns* are not message sends. A return simply causes the computed result to be returned to the sender.
- *Pragmas* are not message sends. They are method annotations.

Other than these few exceptions, pretty much everything else does truly happen by sending messages.

About object-oriented programming.

One of the consequences of Pharo's model of message sending is that it encourages a style in which objects tend to have very small methods and delegate tasks to other objects, rather than implementing huge, procedural methods that assume too much responsibility. Joseph Pelrine expresses this principle succinctly as follows:

> **Note** Don't do anything that you can push off onto someone else.

Many object-oriented languages provide both static and dynamic operations for objects. In Pharo there are only dynamic message sends. For example, instead of providing static class operations, we simply send messages to classes (which are simply objects).

In particular, since there are no *public fields* in Pharo, the only way to update an instance variable of another object is to send it a message asking that it update its own field. Of course, providing setter and getter methods for all the instance variables of an object is not good object-oriented style. Joseph Pelrine also states this very nicely:

> **Note** Don't let anyone else play with your data.

9.7 Sending a message: a two-step process

What exactly happens when an object receives a message? This is a two-step process: *method lookup* and *method execution*.

- **Lookup.** First, the method having the same name as the message is looked up.
- **Method Execution.** Second, the found method is applied to the receiver with the message arguments: When the method is found, the arguments are bound to the parameters of the method, and the virtual machine executes it.

The lookup process is quite simple:

1. The class of the receiver looks up the method to use to handle the message.
2. If this class does not have that method defined, it asks its superclass, and so on, up the inheritance chain.

It is essentially as simple as that. Nevertheless there are a few questions that need some care to answer:

Listing 9-7 A locally implemented method.

```
EllipseMorph >> defaultColor
     "Answer the default color/fill style for the receiver"
     ^ Color yellow
```

Listing 9-8 An inherited method.

```
Morph >> openInWorld
     "Add this morph to the world."
     self openInWorld: self currentWorld
```

- *What happens when a method does not explicitly return a value?*
- *What happens when a class reimplements a superclass method?*
- *What is the difference between* `self` *and* `super` *sends?*
- *What happens when no method is found?*

The rules for method lookup that we present here are conceptual; virtual machine implementors use all kinds of tricks and optimizations to speed up method lookup. That's their job, but you should never be able to detect that they are doing something different from our rules.

First let us look at the basic lookup strategy, and then consider these further questions.

9.8 Method lookup follows the inheritance chain

Suppose we create an instance of `EllipseMorph`.

```
anEllipse := EllipseMorph new.
```

If we send the message `defaultColor` to this object now, we get the result `Color yellow`.

```
anEllipse defaultColor
>>> Color yellow
```

The class `EllipseMorph` implements `defaultColor`, so the appropriate method is found immediately.

In contrast, if we send the message `openInWorld` to `anEllipse`, the method is not immediately found, since the class `EllipseMorph` does not implement `openInWorld`. The search therefore continues in the superclass, `Bordered-Morph`, and so on, until an `openInWorld` method is found in the class `Morph` (see Figure 9-9).

9.9 Method execution

We mentioned that sending a message is a two-step process:

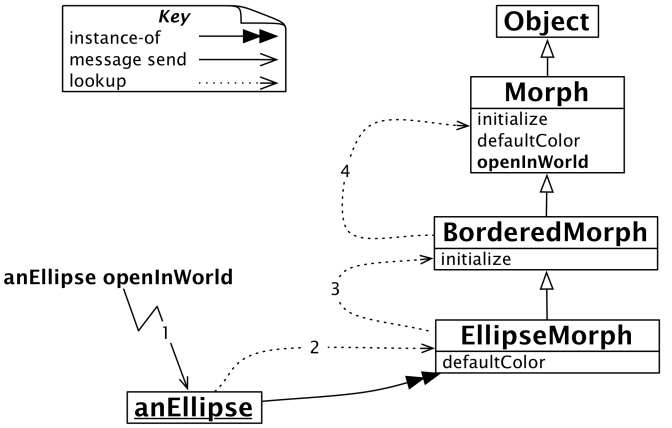

Figure 9-9 Method lookup follows the inheritance hierarchy.

Listing 9-10 Another locally implemented method.

```
EllipseMorph >> closestPointTo: aPoint
  ^ self intersectionWithLineSegmentFromCenterTo: aPoint
```

- **Lookup.** First, the method having the same name as the message is looked up.
- **Method Execution.** Second, the found method is applied to the receiver with the message arguments: When the method is found, the arguments are bound to the parameters of the method, and the virtual machine executes it.

Now we explain the second point: the method execution.

When the lookup returns a method, the receiver of the message is bound to `self`, and the arguments of the message to the method parameters. Then the system executes the method body. This is true wherever the method that should be executed is found. Imagine that we send the message `EllipseMorph new closestPointTo: 100@100` and that the method is defined as in Listing 9-10.

The variable `self` will point to the new ellipse we created and `aPoint` will refer to the point `100@100`.

Now exactly the same process will happen and this even if the method found by the method lookup finds the method in a superclass. When we send the message `EllipseMorph new openInWorld`. The method `openInWorld` is found in the `Morph` class. Still the variable `self` is bound to the newly created ellipse. This is why we say that `self` always represents the receiver of the message and this independently of the class in which the method was found.

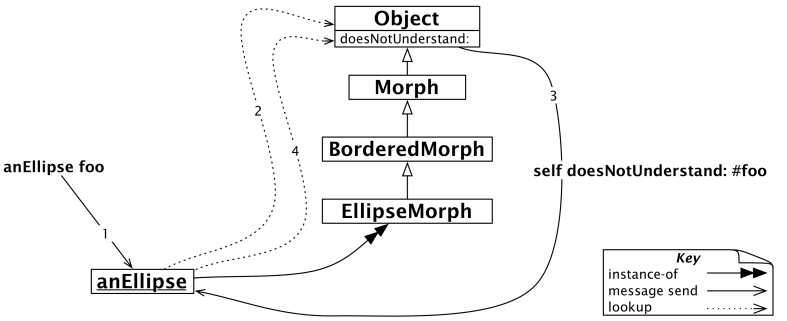

Figure 9-11 Message `foo` is not understood.

This is why there are two different steps during a message send: looking up the method within the class hierarchy of the message receiver and the method execution on the message receiver.

9.10 Message not understood

What happens if the method we are looking for is not found?

Suppose we send the message `foo` to our ellipse. First the normal method lookup will go through the inheritance chain all the way up to `Object` (or rather `ProtoObject`) looking for this method. When this method is not found, the virtual machine will cause the object to send `self doesNotUnderstand: #foo` (See Figure 9-11).

Now, this is a perfectly ordinary, dynamic message send, so the lookup starts again from the class `EllipseMorph`, but this time searching for the method `doesNotUnderstand:`. As it turns out, `Object` implements `doesNotUnderstand:`. This method will create a new `MessageNotUnderstood` object which is capable of starting a Debugger in the current execution context.

Why do we take this convoluted path to handle such an obvious error? Well, this offers developers an easy way to intercept such errors and take alternative action. One could easily override the method `Object>>doesNotUnderstand:` in any subclass of `Object` and provide a different way of handling the error.

In fact, this can be an easy way to implement automatic delegation of messages from one object to another. A `Delegator` object could simply delegate all messages it does not understand to another object whose responsibility it is to handle them, or raise an error itself!

Listing 9-12 Explicitly returning `self`.

```
Morph >> openInWorld
    "Add this morph to the world."
    self openInWorld: self currentWorld
    ^ self
```

9.11 About returning self

Notice that the method `defaultColor` of the class `EllipseMorph` explicitly returns `Color yellow`, whereas the method `openInWorld` of `Morph` does not appear to return anything.

Actually a method *always* answers a message with a value (which is, of course, an object). The answer may be defined by the ^ construct in the method, but if execution reaches the end of the method without executing a ^, the method still answers a value – it answers the object that received the message. We usually say that the method *answers self*, because in Pharo the pseudo-variable `self` represents the receiver of the message, much like the keyword `this` in Java. Other languages, such as Ruby, by default return the value of the last statement in the method. Again, this is not the case in Pharo, instead you can imagine that a method without an explicit return ends with `^ self`.

Important `self` always represents the receiver of the message.

This suggests that `openInWorld` is equivalent to `openInWorldReturnSelf`, defined in Listing 9-12.

Why is explicitly writing `^ self` not a so good thing to do? When you return something explicitly, you are communicating that you are returning something of interest to the sender. When you explicitly return `self`, you are saying that you expect the sender to use the returned value. This is not the case here, so it is best not to explicitly return `self`. We only return `self` on special case to stress that the receiver is returned.

This is a common idiom in Pharo, which Kent Beck refers to as *Interesting return value*: *"Return a value only when you intend for the sender to use the value."*

Important By default (if not specified differently) a method returns the message receiver.

9.12 Overriding and extension

If we look again at the `EllipseMorph` class hierarchy in Figure 9-9, we see that the classes `Morph` and `EllipseMorph` both implement `defaultColor`. In fact, if we open a new morph (`Morph new openInWorld`) we see that we get a blue morph, whereas an ellipse will be yellow by default.

Listing 9-13 Super initialize.

```
BorderedMorph >> initialize
    "Initialize the state of the receiver"

    super initialize.
    self borderInitialize
```

We say that `EllipseMorph` *overrides* the `defaultColor` method that it inherits from `Morph`. The inherited method no longer exists from the point of view of `anEllipse`.

Sometimes we do not want to override inherited methods, but rather *extend* them with some new functionality, that is, we would like to be able to invoke the overridden method *in addition to* the new functionality we are defining in the subclass. In Pharo, as in many object-oriented languages that support single inheritance, this can be done with the help of `super` sends.

A frequent application of this mechanism is in the `initialize` method. Whenever a new instance of a class is initialized, it is critical to also initialize any inherited instance variables. However, the knowledge of how to do this is already captured in the `initialize` methods of each of the superclass in the inheritance chain. The subclass has no business even trying to initialize inherited instance variables!

It is therefore good practice whenever implementing an `initialize` method to send `super initialize` before performing any further initialization as shown in Listing 9-13.

We need `super` sends to compose inherited behaviour that would otherwise be overridden.

> **Important** It is a good practice that an `initialize` method start by sending `super initialize`.

9.13 Self and super sends

`self` represents the receiver of the message and the lookup of the method starts in the class of the receiver. Now what is `super`? `super` is *not* the superclass! It is a common and natural mistake to think this. It is also a mistake to think that lookup starts in the superclass of the class of the receiver.

> **Important** `self` represents the receiver of the message and the method lookup starts in the class of the receiver.

How do `self` sends differ from `super` sends?

Like `self`, `super` represents the receiver of the message. Yes you read it well! The only thing that changes is the method lookup. Instead of lookup

Listing 9-14 A `self` send.

```
Morph >> fullPrintOn: aStream
  aStream nextPutAll: self class name, ' new'
```

Listing 9-15 A `self` send.

```
Morph >> constructorString
    ^ String streamContents: [ :s | self fullPrintOn: s  ].
```

Listing 9-16 Combining `super` and `self` sends.

```
BorderedMorph >> fullPrintOn: aStream
    aStream nextPutAll: '('.
    super fullPrintOn: aStream.
    aStream
        nextPutAll: ') setBorderWidth: ';
        print: borderWidth;
        nextPutAll: ' borderColor: ', (self colorString: borderColor)
```

starting in the class of the receiver, it starts in the *superclass of the class of the method where the* `super` *send occurs*.

> **Important** `super` represents the receiver of the message and the method lookup starts in the superclass of the class of the method where the `super` send occurs.

We shall see in the following example precisely how this works. Imagine that we define the following three methods:

First in Listing 9-14, we define the method `fullPrintOn:` on class `Morph` that just adds to the stream the name of the class followed by the string ' new' - the idea is that we could execute the resulting string and get back an instance similar to the receiver.

Second we define the method `constructorString` that sends the message `fullPrintOn:` (see Listing 9-15).

Finally, we define the method `fullPrintOn:` on the class `BorderedMorph` superclass of `EllipseMorph`. This new method extends the superclass behavior: it invokes it and adds extra behavior (see Listing 9-16).

Consider the message `constructorString` sent to an instance of `EllipseMorph`:

```
EllipseMorph new constructorString
>>> '(EllipseMorph new) setBorderWidth: 1 borderColor: Color black'
```

How exactly is this result obtained through a combination of `self` and `super` sends? First, `anEllipse constructorString` will cause the method `constructorString` to be found in the class `Morph`, as shown in Figure 9-17.

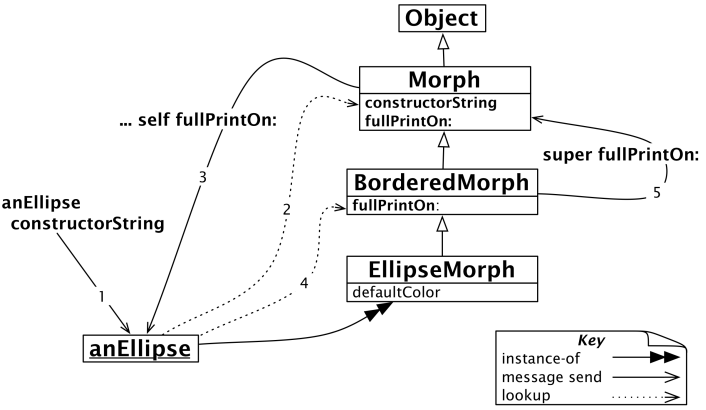

Figure 9-17 `self` and `super` sends.

The method `constructorString` of `Morph` performs a `self` send of `fullPrintOn:`. The message `fullPrintOn:` is looked up starting in the class `EllipseMorph`, and the method `fullPrintOn: BorderedMorph` is found in `BorderedMorph` (see Figure 9-17). What is critical to notice is that the `self` send causes the method lookup to start again in the class of the receiver, namely the class of `anEllipse`.

At this point, the method `fullPrintOn:` of `BorderedMorph` does a `super` send to extend the `fullPrintOn:` behaviour it inherits from its superclass.

Because this is a `super` send, the lookup now starts in the superclass of the class where the `super` send occurs, namely in `Morph`. We then immediately find and execute the method `fullPrintOn:` of the class `Morph`.

9.14 Stepping back

A `self` send is dynamic in the sense that by looking at the method containing it, we cannot predict which method will be executed. Indeed an instance of a subclass may receive the message containing the self expression and redefine the method in that subclass. Here `EllipseMorph` could redefine the method `fullPrintOn:` and this method would be executed by method `constructorString`. Note that by only looking at the method `constructorString`, we cannot predict which `fullPrintOn:` method (either the one of `EllipseMorph`, `BorderedMorph`, or `Morph`) will be executed when executing the method `constructorString`, since it depends on the receiver the `constructorString` message.

> **Important** A `self` send triggers a *method lookup starting in the class of the receiver.* A `self` send is dynamic in the sense that by looking at the method containing it, we cannot predict which method will be executed.

Note that the `super` lookup did not start in the superclass of the receiver. This would have caused lookup to start from `BorderedMorph`, resulting in an infinite loop!

If you think carefully about `super` send and Figure 9-17, you will realize that `super` bindings are static: all that matters is the class in which the text of the `super` send is found. By contrast, the meaning of `self` is dynamic: it always represents the receiver of the currently executing message. This means that *all* messages sent to `self` are looked up by starting in the receiver's class.

> **Important** A `super` send triggers a method lookup starting in the *superclass of the class of the method performing the* `super` *send*. We say that super sends are *static* because just looking at the method we know the class where the lookup should start (the class above the class containing the method).

9.15 The instance and class sides

Since classes are objects, they have their own instance variables and their own methods. We call these *class instance variables* and *class methods*, but they are really no different from ordinary instance variables and methods: They simply operate on different objects (classes in this case). An instance variable describes instance state and a method describes instance behavior. Similarly, class instance variables are just instance variables defined by a metaclass (a class whose instances are classes):

- *Class instance variables* describe the state of classes. An example is the superclass instance variable that describes the superclass of a given class.
- *Class methods* are just methods defined by a metaclass and that will be executed on classes. Sending the message `now` to the class `Date` is defined on the (meta)class `Date class`. This method is executed with the class `Date` as receiver.

A class and its metaclass are two separate classes, even though the former is an instance of the latter. However, this is largely irrelevant to you as a programmer: you are concerned with defining the behavior of your objects and the classes that create them.

For this reason, the browser helps you to browse both class and metaclass as if they were a single thing with two "sides": the *instance side* and the *class side*, as shown in Figure 9-18.

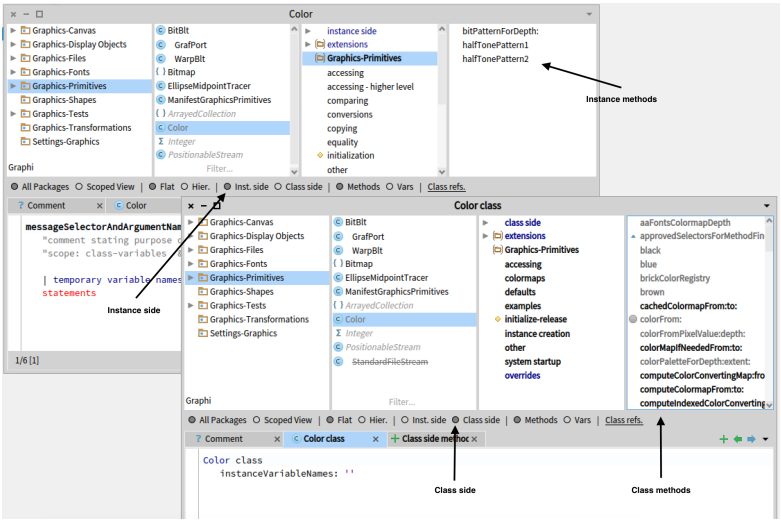

Figure 9-18 Browsing a class and its metaclass.

Listing 9-19 The class method `blue` (defined on the class-side).

```
Color blue
>>> Color blue
"Color instances are self-evaluating"
```

Listing 9-20 Using the accessor method `red` (defined on the instance-side).

```
Color blue red
>>> 0.0
```

- By default, when you select a class in the browser, you're browsing the *instance side* i.e., the methods that are executed when messages are sent to an *instance* of `Color`.
- Clicking on the **Class side** button switches you over to the *class side*: the methods that will be executed when messages are sent to the *class* `Color` itself.

For example, `Color blue` sends the message `blue` to the class `Color`. You will therefore find the method `blue` defined on the *class side* of `Color`, not on the instance side.

Listing 9-21 Using the accessor method `blue` (defined on the instance-side).

```
Color blue blue
>>> 1.0
```

Metaclass creation.

You define a class by filling in the template proposed on the instance side. When you compile this template, the system creates not just the class that you defined, but also the corresponding metaclass (which you can then edit by clicking on the **Class side** button). The only part of the metaclass creation template that makes sense for you to edit directly is the list of the metaclass's instance variable names.

Once a class has been created, browsing its instance side lets you edit and browse the methods that will be possessed by instances of that class (and of its subclasses).

9.16 Class methods

Class methods can be quite useful, you can browse `Color class` for some good examples: You will see that there are two kinds of methods defined on a class: *instance creation methods*, like the class method `blue` in the class `Color class`, and those that perform a utility function, like `Color class>>wheel:`. This is typical, although you will occasionally find class methods used in other ways.

It is convenient to place utility methods on the class side because they can be executed without having to create any additional objects first. Indeed, many of them will contain a comment designed to make it easy to execute them.

Browse method `Color class>>wheel:`, double-click just at the beginning of the comment `"(Color wheel: 12) inspect"` and press `CMD-d`. You will see the effect of executing this method.

For those familiar with Java and C++, class methods may seem similar to static methods. However, the uniformity of the Pharo object model (where classes are just regular objects) means that they are somewhat different: Whereas Java static methods are really just statically-resolved procedures, Pharo class methods are dynamically-dispatched methods. This means that inheritance, overriding and super-sends work for class methods in Pharo, whereas they don't work for static methods in Java.

9.17 Class instance variables

With ordinary *instance* variables, all the instances of a class have the same set of variables (though each instance has its own private set of values), and the instances of its subclasses inherit those variables.

The story is exactly the same with *class* instance variables: a class is an object instance of another class. Therefore the class instance variables are defined on such classes and each class has its own private values for the class instance variables.

Listing 9-22 Dog class definition.

```
Object subclass: #Dog
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Example'
```

Listing 9-23 Adding a class instance variable.

```
Dog class
    instanceVariableNames: 'count'
```

Listing 9-24 Hyena class definition.

```
Dog subclass: #Hyena
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Example'
```

Instance variables also works. Class instance variables are inherited: A subclass will inherit those class instance variables, *but a subclass will have its own private copies of those variables*. Just as objects don't share instance variables, neither do classes and their subclasses share class instance variables.

For example, you could use a class instance variable called `count` to keep track of how many instances you create of a given class. However, any subclass would have its own `count` variable, so subclass instances would be counted separately. The following section presents an example.

9.18 Example: Class instance variables and subclasses

Suppose we define the class `Dog`, and its subclass `Hyena`. Suppose that we add a `count` class instance variable to the class `Dog` (i.e., we define it on the metaclass `Dog class`). `Hyena` will naturally inherit the class instance variable `count` from `Dog`.

Now suppose we define class methods for `Dog` to initialize its `count` to `0`, and to increment it when new instances are created:

Now when we create a new `Dog`, the `count` value of the class `Dog` is incremented, and so is that of the class `Hyena` (but the hyenas are counted separately).

Listing 9-25 Initialize the count of dogs.

```
Dog class >> initialize
    count := 0.
```

Listing 9-26 Keeping count of new dogs.

```
Dog class >> new
    count := count +1.
    ^ super new
```

Listing 9-27 Accessing to count.

```
Dog class >> count
    ^ count
```

```
Dog initialize.
Hyena initialize.
Dog count
>>> 0
```

About class `initialize`.

When you instantiate an object such as `Dog new`, `initialize` is called automatically as part of the `new` message send (you can see for yourself by browsing the method `new` in the class `Behavior`). But with classes, simply defining them does not automatically call `initialize` because it is not clear to the system if a class is fully working. So we have to call `initialize` explicitly here. By default class `initialize` methods are automatically executed only when classes are loaded. See also the discussion about lazy initialization, below.

```
Hyena count
>>> 0
```

```
| aDog |
aDog := Dog new.
Dog count
>>> 1  "Incremented"
```

```
Hyena count
>>> 0  "Still the same"
```

9.19 Stepping back

Class instance variables are private to a class in exactly the same way that instance variables are private to an instance. Since classes and their instances are different objects, this has the following consequences:

1. A class does not have access to the instance variables of its own instances. So, the class `Color` does not have access to the variables of an object instantiated from it, `aColorRed`. In other words, just because a class was used to create an instance (using `new` or a helper instance creation method like `Color red`), it doesn't give *the class* any special direct access to that instance's vari-

ables. The class instead has to go through the accessor methods (a public interface) just like any other object.

2. The reverse is also true: an *instance* of a class does not have access to the class instance variables of its class. In our example above, `aDog` (an individual instance) does not have direct access to the `count` variable of the `Dog` class (except, again, through an accessor method).

> **Important** A class does not have access to the instance variables of its own instances. An instance of a class does not have access to the class instance variables of its class.

For this reason, instance initialization methods must always be defined on the instance side, the class side has no access to instance variables, and so cannot initialize them! All that the class can do is to send initialization messages, using accessors, to newly created instances.

Java has nothing equivalent to class instance variables. Java and C++ static variables are more like Pharo class variables (discussed in Section 9.22), since in those languages all of the subclasses and all of their instances share the same static variable.

9.20 Example: Defining a Singleton

Singleton is the most misunderstood design pattern. When wrongly applied, it favors procedural style promoting single global access. However, the Singleton pattern provides a typical example of the use of class instance variables and class methods.

Imagine that we would like to implement a class `WebServer`, and to use the Singleton pattern to ensure that it has only one instance.

We define the class `WebServer` as follow.

```
Object subclass: #WebServer
    instanceVariableNames: 'sessions'
    classVariableNames: ''
    package: 'Web'
```

Then, clicking on the **Class side** button, we add the (class) instance variable `uniqueInstance`.

```
WebServer class
    instanceVariableNames: 'uniqueInstance'
```

As a result, the class `WebServer class` will have a new instance variable (in addition to the variables that it inherits from `Behavior`, such as `superclass` and `methodDict`). It means that the value of this extra instance variable will describe the instance of the class `WebServer class` i.e., the class `WebServer`.

Listing 9-29 New state for classes.

```
WebServer class allInstVarNames
>>>  "#(#superclass #methodDict #format #layout #organization
    #subclasses #name #classPool #sharedPools #environment #category
    #uniqueInstance)"
```

Listing 9-30 Class-side accessor method `uniqueInstance`.

```
WebServer class >> uniqueInstance
     uniqueInstance ifNil: [ uniqueInstance := self new ].
     ^ uniqueInstance
```

```
Point class allInstVarNames
>>>  "#(#superclass #methodDict #format #layout #organization
    #subclasses #name #classPool #sharedPools #environment
    #category)"
```

We can now define a class method named `uniqueInstance`, as shown below. This method first checks whether `uniqueInstance` has been initialized. If it has not, the method creates an instance and assigns it to the class instance variable `uniqueInstance`. Finally the value of `uniqueInstance` is returned. Since `uniqueInstance` is a class instance variable, this method can directly access it.

The first time that `WebServer uniqueInstance` is executed, an instance of the class `WebServer` will be created and assigned to the `uniqueInstance` variable. The next time, the previously created instance will be returned instead of creating a new one. (This pattern, checking if a variable is nil in an accessor method, and initializing its value if it is nil, is called *lazy initialization*).

Note that the instance creation code in the code above. Script 9-30 is written as `self new` and not as `WebServer new`. What is the difference? Since the `uniqueInstance` method is defined in `WebServer class`, you might think that there is no difference. And indeed, until someone creates a subclass of `WebServer`, they are the same. But suppose that `ReliableWebServer` is a subclass of `WebServer`, and inherits the `uniqueInstance` method. We would clearly expect `ReliableWebServer uniqueInstance` to answer a `ReliableWebServer`. Using `self` ensures that this will happen, since `self` will be bound to the respective receiver, here the classes `WebServer` and `ReliableWebServer`. Note also that `WebServer` and `ReliableWebServer` will each have a different value for their `uniqueInstance` instance variable.

A note on lazy initialization.

Do not over-use the lazy initialization pattern. The setting of initial values for instances of objects generally belongs in the `initialize` method. Putting initialization calls only in `initialize` helps from a readability perspective

– you don't have to hunt through all the accessor methods to see what the initial values are. Although it may be tempting to instead initialize instance variables in their respective accessor methods (using `ifNil:` checks), avoid this unless you have a good reason.

For example, in our `uniqueInstance` method above, we used lazy initialization because users won't typically expect to call `WebServer initialize`. Instead, they expect the class to be "ready" to return new unique instances. Because of this, lazy initialization makes sense. Similarly, if a variable is expensive to initialize (opening a database connection or a network socket, for example), you will sometimes choose to delay that initialization until you actually need it.

9.21 Shared variables

Now we will look at an aspect of Pharo that is not so easily covered by our five rules: shared variables.

Pharo provides three kinds of shared variables:

1. *Globally* shared variables.

2. *Class variables*: variables shared between instances and classes. (Not to be confused with class instance variables, discussed earlier).

3. *Pool variables*: variables shared amongst a group of classes,

The names of all of these shared variables start with a capital letter, to warn us that they are indeed shared between multiple objects.

Global variables

In Pharo, all global variables are stored in a namespace called `Smalltalk globals`, which is implemented as an instance of the class `SystemDictionary`. Global variables are accessible everywhere. Every class is named by a global variable. In addition, a few globals are used to name special or commonly useful objects.

The variable `Processor` names an instance of `ProcessScheduler`, the main process scheduler of Pharo.

```
Processor class
>>> ProcessorScheduler
```

Other useful global variables

`Smalltalk` is the instance of `SmalltalkImage`. It contains many functionality to manage the system. In particular it holds a reference to the main namespace `Smalltalk globals`. This namespace includes `Smalltalk` itself

since it is a global variable. The keys to this namespace are the symbols that name the global objects in Pharo code. So, for example:

```
Smalltalk globals at: #Boolean
>>> Boolean
```

Since `Smalltalk` is itself a global variable:

```
Smalltalk globals at: #Smalltalk
>>> Smalltalk
```

```
(Smalltalk globals at: #Smalltalk) == Smalltalk
>>> true
```

`World` is an instance of `PasteUpMorph` that represents the screen. `World bounds` answers a rectangle that defines the whole screen space; all Morphs on the screen are submorphs of `World`.

`Undeclared` is another dictionary, which contains all the undeclared variables. If you write a method that references an undeclared variable, the browser will normally prompt you to declare it, for example as a global or as an instance variable of the class. However, if you later delete the declaration, the code will then reference an undeclared variable. Inspecting `Undeclared` can sometimes help explain strange behaviour!

Using globals in your code

The recommended practice is to strictly limit the use of global variables. It is usually better to use class instance variables or class variables, and to provide class methods to access them. Indeed, if Pharo were to be implemented from scratch today, most of the global variables that are not classes would be replaced by singletons or others.

The usual way to define a global is just to perform `Do it` on an assignment to a capitalized but undeclared identifier. The parser will then offer to declare the global for you. If you want to define a global programmatically, just execute `Smalltalk globals at: #AGlobalName put: nil`. To remove it, execute `Smalltalk globals removeKey: #AGlobalName`.

9.22 Class variables: Shared variables

Sometimes we need to share some data amongst all the instances of a class and the class itself. This is possible using *class variables*. The term *class variable* indicates that the lifetime of the variable is the same as that of the class. However, what the term does not convey is that these variables are shared amongst all the instances of a class as well as the class itself, as shown in Figure 9-31. Indeed, a better name would have been *shared variables* since this expresses more clearly their role, and also warns of the danger of using them, particularly if they are modified.

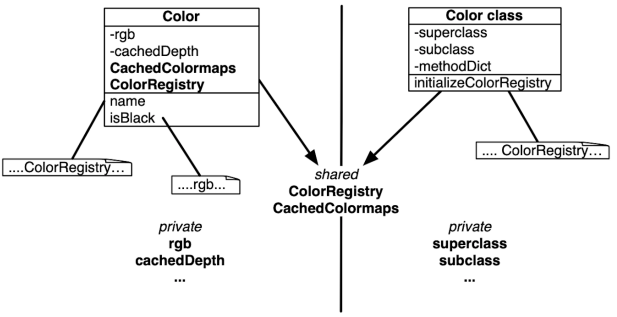

Figure 9-31 Instance and class methods accessing different variables.

Listing 9-32 Color and its class variables.

```
Object subclass: #Color
  instanceVariableNames: 'rgb cachedDepth cachedBitPattern alpha'
  classVariableNames: 'BlueShift CachedColormaps ColorRegistry
    ComponentMask ComponentMax GrayToIndexMap GreenShift
    HalfComponentMask IndexedColors MaskingMap RedShift'
  package: 'Colors-Base'
```

In Figure 9-31 we see that `rgb` and `cachedDepth` are instance variables of `Color`, hence only accessible to instances of `Color`. We also see that `superclass`, `subclass`, `methodDict` and so on are *class instance variables*, i.e., instance variables only accessible to the `Color` class.

But we can also see something new: `ColorRegistry` and `CachedColormaps` are *class variables* defined for `Color`. The capitalization of these variables gives us a hint that they are shared. In fact, not only may all instances of `Color` access these shared variables, but also the `Color` class itself, *and any of its subclasses*. Both instance methods and class methods can access these shared variables.

A class variable is declared in the class definition template. For example, the class `Color` defines a large number of class variables to speed up color creation; its definition is shown below in Script 9-32.

The class variable `ColorRegistry` is an instance of `IdentityDictionary` containing the frequently-used colors, referenced by name. This dictionary is shared by all the instances of `Color`, as well as the class itself. It is accessible from all the instance and class methods.

Listing 9-33 Using Lazy initialization.

```
   ColorNames ifNil: [ self initializeNames ].
   ^ ColorNames
```

Listing 9-34 Initializing the `Color` class.

```
Color class >> initialize
    ...
    self initializeColorRegistry.
    ...
```

Class initialization

The presence of class variables raises the question: how do we initialize them?

One solution is lazy initialization (discussed earlier in this chapter). This can be done by introducing an accessor method which, when executed, initializes the variable if it has not yet been initialized. This implies that we must use the accessor all the time and never use the class variable directly. This furthermore imposes the cost of the accessor send and the initialization test.

Another solution is to override the class method `initialize` (we've seen this before in the Dog example).

If you adopt this solution, you will need to remember to invoke the `initialize` method after you define it (by evaluating `Color initialize`). Although class side `initialize` methods are executed automatically when code is loaded into memory (from a Monticello repository, for example), they are *not* executed automatically when they are first typed into the browser and compiled, or when they are edited and re-compiled.

9.23 Pool variables

Pool variables are variables that are shared between several classes that may not be related by inheritance. Pool variables should be defined as class variables of dedicated classes (subclasses of `SharedPool` as shown below). Our advice is to avoid them; you will need them only in rare and specific circumstances. Our goal here is therefore to explain pool variables just enough so that you can understand them when you are reading code.

A class that accesses a pool variable must mention the pool in its class definition. For example, the class `Text` indicates that it is using the pool `TextConstants`, which contains all the text constants such as `CR` and `LF`. `TextConstants` defines the variables `CR` that is bound to the value `Character cr`, i.e., the carriage return character.

This allows methods of the class `Text` to access the variables of the shared pool in the method body *directly*. For example, we can write the following

Listing 9-35 Pool dictionaries in the `Text` class.

```
ArrayedCollection subclass: #Text
        instanceVariableNames: 'string runs'
        classVariableNames: ''
        poolDictionaries: 'TextConstants'
        package: 'Collections-Text'
```

Listing 9-36 `Text>>testCR`.

```
Text >> testCR
    ^ CR == Character cr
```

method. We see that eventhough `Text` does not define a variable `CR`, since it declared that it uses the shared pool `TextConstants`, it can directly access it.

Here is how `TextConstants` is created. `TextConstants` is a special class subclass of `SharedPool` and it holds class variables.

```
SharedPool subclass: #TextConstants
  instanceVariableNames: ''
  classVariableNames: 'BS BS2 Basal Bold CR Centered Clear CrossedX
    CtrlA CtrlB CtrlC
  CtrlD CtrlDigits CtrlE CtrlF CtrlG CtrlH
  CtrlI CtrlJ CtrlK CtrlL CtrlM CtrlN CtrlO CtrlOpenBrackets CtrlP
    CtrlQ CtrlR CtrlS CtrlT
  CtrlU CtrlV CtrlW CtrlX CtrlY CtrlZ Ctrla Ctrlb Ctrlc Ctrld Ctrle
    Ctrlf Ctrlg Ctrlh Ctrli
  Ctrlj Ctrlk Ctrll Ctrlm Ctrln Ctrlo Ctrlp Ctrlq Ctrlr Ctrls Ctrlt
    Ctrlu Ctrlv Ctrlw
  Ctrlx Ctrly Ctrlz DefaultBaseline DefaultFontFamilySize
    DefaultLineGrid DefaultMarginTabsArray
  DefaultMask DefaultRule DefaultSpace DefaultTab DefaultTabsArray
    ESC EndOfRun Enter Italic
  Justified LeftFlush LeftMarginTab RightFlush RightMarginTab Space
    Tab TextSharedInformation'
  package: 'Text-Core-Base'
```

Once again, we recommend that you avoid the use of pool variables and pool dictionaries.

9.24 Abstract methods and abstract classes

An abstract class is a class that exists to be subclassed, rather than to be instantiated. An abstract class is usually incomplete, in the sense that it does not define all of the methods that it uses. The ”placeholder” methods, those that the other methods assume to be (re)defined are called abstract methods.

Pharo has no dedicated syntax to specify that a method or a class is abstract. Instead, by convention, the body of an abstract method consists of the expression `self subclassResponsibility`. This indicates that subclasses

Listing 9-37 `Magnitude>> <.`

```
< aMagnitude
    "Answer whether the receiver is less than the argument."

    ^self subclassResponsibility
```

Listing 9-38 `Magnitude>> >=.`

```
>= aMagnitude
    "Answer whether the receiver is greater than or equal to the
    argument."

    ^(self < aMagnitude) not
```

Listing 9-39 `Character>> <=.`

```
< aCharacter
    "Answer true if the receiver's value < aCharacter's value."

    ^self asciiValue < aCharacter asciiValue
```

have the responsibility to define a concrete version of the method. `self subclassResponsibility` methods should always be overridden, and thus should never be executed. If you forget to override one, and it is executed, an exception will be raised.

Similarly, a class is considered abstract if one of its methods is abstract. Nothing actually prevents you from creating an instance of an abstract class; everything will work until an abstract method is invoked.

Example: the abstract class `Magnitude`

`Magnitude` is an abstract class that helps us to define objects that can be compared to each other. Subclasses of `Magnitude` should implement the methods <, = and `hash`. Using such messages, `Magnitude` defines other methods such as >, >=, <=, `max:`, `min: between:and:` and others for comparing objects. Such methods are inherited by subclasses. The method `Magnitude>><` is abstract, and defined as shown in the following script.

By contrast, the method >= is concrete, and is defined in terms of <.

The same is true of the other comparison methods (they are all defined in terms of the abstract method <).

`Character` is a subclass of `Magnitude`; it overrides the < method (which, if you recall, is marked as abstract in `Magnitude` by the use of `self subclassResponsibility`) with its own version (see the method definition below).

`Character` also explicitly defines methods = and `hash`; it inherits from `Magnitude` the methods >=, <=, ~= and others.

9.25 Chapter summary

The object model of Pharo is both simple and uniform. Everything is an object, and pretty much everything happens by sending messages.

- Everything is an object. Primitive entities like integers are objects, but also classes are first-class objects.
- Every object is an instance of a class. Classes define the structure of their instances via *private* instance variables and the behaviour of their instances via *public* methods. Each class is the unique instance of its metaclass. Class variables are private variables shared by the class and all the instances of the class. Classes cannot directly access instance variables of their instances, and instances cannot access instance variables of their class. Accessors must be defined if this is needed.
- Every class has a superclass. The root of the single inheritance hierarchy is `ProtoObject`. Classes you define, however, should normally inherit from `Object` or its subclasses. There is no syntax for defining abstract classes. An abstract class is simply a class with an abstract method (one whose implementation consists of the expression `self subclassResponsibility`). Although Pharo supports only single inheritance, it is easy to share implementations of methods by packaging them as *traits*.
- Everything happens by sending messages. We do not *call methods*, we *send messages*. The receiver then chooses its own method for responding to the message.
- Method lookup follows the inheritance chain; `self` sends are dynamic and start the method lookup in the class of the receiver, whereas `super` sends start the method lookup in the superclass of class in which the `super` send is written. From this perspective `super` sends are more static than `self` sends.
- There are three kinds of shared variables. Global variables are accessible everywhere in the system. Class variables are shared between a class, its subclasses and its instances. Pool variables are shared between a selected set of classes. You should avoid shared variables as much as possible.

CHAPTER 10

Traits: reusable class fragments

Although Pharo does not provide multiple inheritance, it supports a mechanism called Traits for sharing class fragments (behaviour and state) across unrelated classes. Traits in their simpler form are just collections of methods that can be reused by multiple classes that are not related by inheritance. Since Pharo 7.0, traits can also hold state. Using traits allows one to share code between different classes without duplicating code. This makes it easy for classes to have a unique superclass, yet still reuse useful behavior with otherwise unrelated classes.

As we will show later, traits propose a way to compose and resolve conflicts in disciplined manner (and not just expecting that the latest loaded method is the correct one as this happens with language such as Groovy). The basic principle is that the composer (be it a class or a trait) takes always precedence and can decide in its context how to resolve a conflict. Methods can be removed or accessible under a new name at composition time.

10.1 A simple trait

The following 10-1 define a trait. The `uses:` clauses in an empty array indicating that this trait is not composed of other traits.

The trait `TFlyingAbility` defines a single method `fly`.

Listing 10-1 A simple trait.

```
Trait named: #TFlyingAbility
   uses: {}
   package: 'Traits-Example'
```

```
TFlyingAbility >> fly
  ^ 'I''m flying!'
```

Now we define a class called `Bird` that uses the trait. It means that the class has a method called `fly`.

```
Object subclass: #Bird
  uses: TFlyingAbility
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Instances of the class `Bird` know how to answer to the message `fly`.

```
| b |
b := Bird new.
b fly
>>> 'I''m flying!'
```

Using a required method

A trait can invoke methods that are available on the class using it. Here the method `greeting` of the trait `TGreetable` in invoking the method `name` that is not defined in the trait itself. In such a case the class using the trait will have to implement such *required* method.

```
Trait named: #TGreetable
   uses: {}
   package: 'Traits-Example'
```

```
TGreetable >> greeting
  ^ 'Hello ', self name
```

Notice that `self` in a trait represents the receiver of the message. Nothing changes.

```
Object subclass: #Person
  uses: TGreetable
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Now in the class `Person` we define the method `name` and the `greeting` method will invoke it.

```
Person >> name
  ^ 'Bob'
```

```
Person new greeting
>>> 'Hello Bob'
```

10.2 Self in a trait is the receiver

The question of the status of `self` in a trait may be raised. However, there is not difference. `self` represents the receiver. The fact that the method is defined in a class or a trait has no influence on `self`.

```
Trait named: #TInspector
  uses: {}
  package: 'Traits-Example'
```

```
TInspector >> whoAmI
  ^ self
```

```
Object subclass: #Foo
  uses: TInspector
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

```
| foo |
foo := Foo new.
foo whoAmI == foo
>>> true
```

10.3 Trait state

Since Pharo 7.0 traits can also define instance variables.

```
Trait named: #TCounting
   uses: {}
   slots: { #count }
   package: 'Traits-Example'
```

```
TCounting >> initializeTCounting
  count := 0
```

```
TCounting >> increment
  count := count + 1.
  ^ count
```

```
Object subclass: #Counter
  uses: TCounting
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

```
Counter >> initialize
  self initializeTCounting
```

```
Counter new increment; increment
>>> 2
```

10.4 A class can use two traits

```
Trait named: #TSpeakingAbility
   uses: {}
   package: 'Traits-Example'
```

```
TSpeakingAbility >> speak
  ^ 'I''m speaking!'
```

```
Trait named: #TSpeakingAbility
   uses: {}
   package: 'Traits-Example'
```

```
Object subclass: #Duck
  uses: TFlyingAbility + TSpeakingAbility
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

```
| d |
d := Duck new.
d speak
>>> 'I''m speaking!'
d fly
>>> 'I''m flying!'
```

10.5 Overriding method take always precedence over traits

A method originating from a trait acts as if it would have been defined in the class itself. Now the user of a trait (be it a class or another trait) can always redefine the method originating from the trait and the redefinition takes precedence in the user over the method of trait.

In the class `Duck` we can redefine the method `speak` to do something else and for example send the message `quack`.

```
Duck >> quack
  ^ 'QUACK'
```

```
Duck >> speak
  ^ self quack
```

```
| d |
d := Duck new.
d speak
>>> 'QUACK'
```

10.6 Composing a trait out of other traits

Trait composition expressions may combine multiple traits using the + operator. In case of conflicts (i.e., if multiple traits define methods with the same name), these conflicts can be resolved by explicitly removing these methods (with -), or by redefining these methods in the class or trait that you are defining. It is also possible to *alias* methods (with @), providing a new name for them.

10.7 Managing conflicts

10.8 Conclusion

CHAPTER 11

SUnit: Tests in Pharo

SUnit is a minimal yet powerful framework that supports the creation and deployment of tests. As might be guessed from its name, the design of SUnit focussed on *Unit Tests*, but in fact it can be used for integration tests and functional tests as well. SUnit was originally developed by Kent Beck and subsequently extended by Joseph Pelrine and others to incorporate the notion of a resource (discussed below).

This chapter is as short as possible to show you that tests are simple. For a more in depth description of SUnit and different approaches of testing you can read the book: *Testing in Pharo* available at http://books.pharo.org.

In this chapter we start by discussing why we test, and what makes a good test. We then present a series of small examples showing how to use SUnit. Finally, we look at the implementation of SUnit, so that you can understand how Pharo uses the power of reflection in supporting its tools. Note that the version documented in this chapter and used in Pharo is a modified version of SUnit3.3.

11.1 Introduction

The interest in testing and Test Driven Development is not limited to Pharo. Automated testing has become a hallmark of the *Agile software development* movement, and any software developer concerned with improving software quality would do well to adopt it. Indeed, developers in many languages have come to appreciate the power of unit testing, and versions of *xUnit* now exist for every programming language.

Neither testing, nor the building of test suites, is new. By now, everybody knows that tests are a good way to catch errors. eXtreme Programming,

by making testing a core practice and by emphasizing *automated* tests, has helped to make testing productive and fun, rather than a chore that programmers dislike. The Pharo community has a long tradition of testing because of the incremental style of development supported by its programming environment. In traditional Pharo development, the programmer would write tests in a playground as soon as a method was finished. Sometimes a test would be incorporated as a comment at the head of the method that it exercised, or tests that needed some set up would be included as example methods in the class. The problem with these practices is that tests in a playground are not available to other programmers who modify the code. Comments and example methods are better in this respect, but there is still no easy way to keep track of them and to run them automatically. Tests that are not run do not help you to find bugs! Moreover, an example method does not inform the reader of the expected result: you can run the example and see the (perhaps surprising) result, but you will not know if the observed behaviour is correct.

SUnit is valuable because it allows us to write tests that are self-checking: the test itself defines what the correct result should be. It also helps us to organize tests into groups, to describe the context in which the tests must run, and to run a group of tests automatically. In less than two minutes you can write tests using SUnit, so instead of writing small code snippets in a playground, we encourage you to use SUnit and get all the advantages of stored and automatically executable tests.

11.2 Why testing is important

Unfortunately, many developers believe that tests are a waste of their time. After all, *they* do not write bugs, only *other* programmers do that. Most of us have said, at some time or other: *I would write tests if I had more time.* If you never write a bug, and if your code will never be changed in the future, then indeed tests are a waste of your time. However, this most likely also means that your application is trivial, or that it is not used by you or anyone else. Think of tests as an investment for the future: having a suite of tests is quite useful now, but it will be *extremely* useful when your application, or the environment in which it runs, changes in the future.

Tests play several roles. First, they provide documentation of the functionality that they cover. This documentation is active: watching the tests pass tells you that the documentation is up to date. Second, tests help developers to confirm that some changes that they have just made to a package have not broken anything else in the system, and to find the parts that break when that confidence turns out to be misplaced. Finally, writing tests during, or even before, programming forces you to think about the functionality that you want to design, *and how it should appear to the client code*, rather than about how to implement it.

By writing the tests first, i.e., before the code, you are compelled to state the context in which your functionality will run, the way it will interact with the client code, and the expected results. Your code will improve. Try it.

We cannot test all aspects of any realistic application. Covering a complete application is simply impossible and should not be the goal of testing. Even with a good test suite some bugs will still creep into the application, where they can lay dormant waiting for an opportunity to damage your system. If you find that this has happened, take advantage of it! As soon as you uncover the bug, write a test that exposes it, run the test, and watch it fail. Now you can start to fix the bug: the tests will tell you when you are done.

11.3 What makes a good test?

Writing good tests is a skill that can be learned by practicing. Let us look at the properties that tests should have to get the maximum benefit.

Tests should be repeatable. You should be able to run a test as often as you want, and always get the same answer.

Tests should run without human intervention. You should be able to run them unattended.

Tests should tell a story. Each test should cover one aspect of a piece of code. A test should act as a scenario that you or someone else can read to understand a piece of functionality.

Tests should have a change frequency lower than that of the functionality they cover. You do not want to have to change all your tests every time you modify your application. One way to achieve this is to write tests based on the public interfaces of the class that you are testing. It is OK to write a test for a private *helper* method if you feel that the method is complicated enough to need the test, but you should be aware that such a test may have to be changed, or thrown away entirely, when you think of a better implementation.

One consequence of such properties is that the number of tests should be somewhat proportional to the number of functions to be tested: changing one aspect of the system should not break all the tests but only a limited number. This is important because having 100 tests fail should send a much stronger message than having 10 tests fail. However, it is not always possible to achieve this ideal: in particular, if a change breaks the initialization of an object, or the set-up of a test, it is likely to cause all of the tests to fail.

Several software development methodologies such as *eXtreme Programming* and Test-Driven Development (TDD) advocate writing tests before writing code. This may seem to go against our deep instincts as software developers. All we can say is: go ahead and try it. We have found that writing the tests before the code helps us to know what we want to code, helps us know when we are done, and helps us conceptualize the functionality of a class and to

Listing 11-1 An Example Set Test class

```
TestCase subclass: #MyExampleSetTest
  instanceVariableNames: 'full empty'
  classVariableNames: ''
  package: 'MySetTest'
```

design its interface. Moreover, test-first development gives us the courage to go fast, because we are not afraid that we will forget something important.

Writing tests is not difficult in itself. Now let's write our first test, and show you the benefits of using SUnit.

SUnit by example

Before going into the details of SUnit, we will show a step by step example. We use an example that tests the class `Set`. Try entering the code as we go along.

11.4 Step 1: Create the test class

First you should create a new subclass of `TestCase` called `MyExampleSetTest`. Add two instance variables so that your new class looks like this:

We will use the class `MyExampleSetTest` to group all the tests related to the class `Set`. It defines the context in which the tests will run. Here the context is described by the two instance variables `full` and `empty` that we will use to represent a full and an empty set.

The name of the class is not critical, but by convention it should end in `Test`. If you define a class called `Pattern` and call the corresponding test class `PatternTest`, the two classes will be alphabetized together in the browser (assuming that they are in the same package). It *is* critical that your class is a subclass of `TestCase`.

11.5 Step 2: Initialize the test context

The message `TestCase >> setUp` defines the context in which the tests will run, a bit like an initialize method. `setUp` is invoked before the execution of each test method defined in the test class.

Define the `setUp` method as follows, to initialize the `empty` variable to refer to an empty set and the `full` variable to refer to a set containing two elements.

```
MyExampleSetTest >> setUp
  empty := Set new.
  full := Set with: 5 with: 6
```

In testing jargon the context is called the *fixture* for the test.

11.6 Step 3: Write some test methods

Let's create some tests by defining some methods in the class `MyExampleSetTest`. Each method represents one test. The names of the methods should start with the string `'test'` so that SUnit will collect them into test suites. Test methods take no arguments.

Define the following test methods. The first test, named `testIncludes`, tests the `includes:` method of `Set`. The test says that sending the message `includes: 5` to a set containing 5 should return `true`. Clearly, this test relies on the fact that the `setUp` method has already run.

```
MyExampleSetTest >> testIncludes
  self assert: (full includes: 5).
  self assert: (full includes: 6)
```

The second test, named `testOccurrences`, verifies that the number of occurrences of 5 in `full` set is equal to one, even if we add another element 5 to the set.

```
MyExampleSetTest >> testOccurrences
  self assert: (empty occurrencesOf: 0) equals: 0.
  self assert: (full occurrencesOf: 5) equals: 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) equals: 1
```

Finally, we test that the set no longer contains the element 5 after we have removed it.

```
MyExampleSetTest >> testRemove
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Note the use of the method `TestCase >> deny:` to assert something that should not be true. `aTest deny: anExpression` is equivalent to `aTest assert: anExpression not`, but is much more readable.

11.7 Step 4: Run the tests

The easiest way to run the tests is directly from the browser. Simply click on the icon of the class name, or on an individual test method, and select *Run tests (t)* or press the icon. The test methods will be flagged green or red, depending on whether they pass or not (as shown in 11-2).

You can also select sets of test suites to run, and obtain a more detailed log of the results using the SUnit Test Runner, which you can open by selecting `World > Test Runner`.

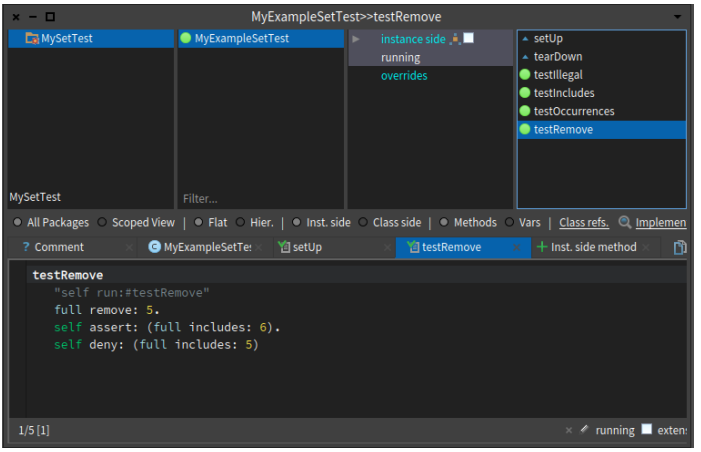

Figure 11-2 Running SUnit tests from the System Browser.

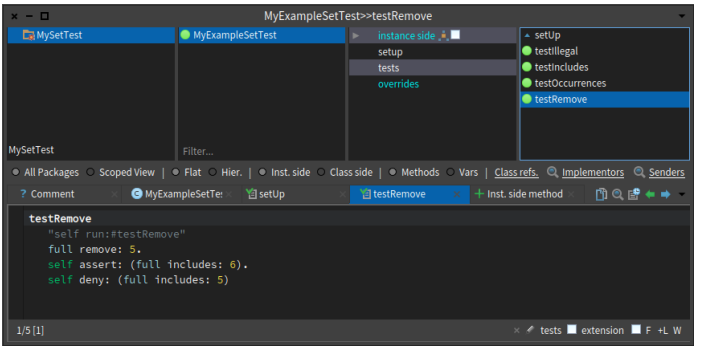

Figure 11-3 Running SUnit tests using the *TestRunner*.

The *Test Runner*, shown in Figure 11-3, is designed to make it easy to execute groups of tests.

The left-most pane lists all of the packages that contain test classes (i.e., subclasses of `TestCase`). When some of these packages are selected, the test classes that they contain appear in the pane to the right. Abstract classes are italicized, and the test class hierarchy is shown by indentation, so subclasses of `ClassTestCase` are indented more than subclasses of `TestCase`. `ClassTestCase` is a class offering utilities methods to compute test coverage.

Open a Test Runner, select the package *MySetTest*, and click the `Run Selected` button.

You can also run a single test (and print the usual pass/fail result summary)

by executing a *Print it* on the following code: `MyExampleSetTest run: #testRemove`.

Some people include an executable comment in their test methods that allows running a test method with a *Do it* from the browser, as shown below.

```
MyExampleSetTest >> testRemove
  "self run: #testRemove"
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Introduce a bug in `MyExampleSetTest >> testRemove` and run the tests again. For example, change `6` to `7`, as in:

```
MyExampleSetTest >> testRemove
  full remove: 5.
  self assert: (full includes: 7).
  self deny: (full includes: 5)
```

The tests that did not pass (if any) are listed in the right-hand panes of the *Test Runner*. If you want to debug one, to see why it failed, just click on the name. Alternatively, you can execute one of the following expressions:

```
(MyExampleSetTest selector: #testRemove) debug

MyExampleSetTest debug: #testRemove
```

11.8 Step 5: Interpret the results

The method `assert:` is defined in the class `TestAsserter`. This is a superclass of `TestCase` and therefore all other `TestCase` subclasses and is responsible for all kind of test result assertions. The `assert:` method expects a boolean argument, usually the value of a tested expression. When the argument is true, the test passes; when the argument is false, the test fails.

There are actually three possible outcomes of a test: *passing*, *failing*, and *raising an error*.

- **Passing**. The outcome that we hope for is that all of the assertions in the test are true, in which case the test passes. In the test runner, when all of the tests pass, the bar at the top turns green. However, there are two other ways that running a test can go wrong.
- **Failing**. The obvious way is that one of the assertions can be false, causing the test to *fail*.
- **Error**. The other possibility is that some kind of error occurs during the execution of the test, such as a *message not understood* error or an *index out of bounds* error. If an error occurs, the assertions in the test method may not have been executed at all, so we can't say that the test has failed; nevertheless, something is clearly wrong!

In the *test runner*, failing tests cause the bar at the top to turn yellow, and are listed in the middle pane on the right, whereas tests with errors cause the bar to turn red, and are listed in the bottom pane on the right.

Modify your tests to provoke both errors and failures.

11.9 The SUnit cookbook

This section will give you more details on how to use SUnit. If you have used another testing framework such as JUnit, much of this will be familiar, since all these frameworks have their roots in SUnit. Normally you will use SUnit's GUI to run tests, but there are situations where you may not want to use it.

Using `assert:equals:`

`assert:equals:` that offers a better report than `assert:` in case of error. For example, the two following tests are equivalent. However, the second one will report the value that the test is expecting: this makes easier to understand the failure. In this example, we suppose that `aDateAndTime` is an instance variable of the test class.

```
testAsDate
  self assert: aDateAndTime asDate = ('February 29, 2004' asDate
    translateTo: 2 hours).
```

```
testAsDate
  self
    assert: aDateAndTime asDate
    equals: ('February 29, 2004' asDate translateTo: 2 hours).
```

Skipping a test

Sometimes in the middle of a development, you may want to skip a test instead of removing it or renaming it to prevent it from running. You can simply invoke the `TestAsserter` message `skip` on your test case instance. For example, the following test uses it to define a conditional test.

```
OCCompiledMethodIntegrityTest >> testPragmas

  | newCompiledMethod originalCompiledMethod |
  (Smalltalk globals hasClassNamed: #Compiler) ifFalse: [ ^ self
    skip ].
  ...
```

This is handy to make sure that your automated execution of tests is reporting success.

Listing 11-4 Testing error raising

```
MyExampleSetTest >> testIllegal
  self should: [ empty at: 5 ] raise: Error.
  self should: [ empty at: 5 put: #zork ] raise: Error
```

Other assertions

In addition to `assert:` and `deny:`, there are several other methods that can be used to make assertions.

First, `TestAsserter >> assert:description:` and `TestAsserter >> deny:description:` take a second argument which is a message string that describes the reason for the failure, if it is not obvious from the test itself. These methods are described in Section **??**.

Next, SUnit provides two additional methods, `TestAsserter >> should:raise:` and `TestAsserter >> shouldnt:raise:` for testing exception propagation.

For example, you would use `self should: aBlock raise: anException` to test that a particular exception is raised during the execution of `aBlock`. The method below illustrates the use of `should:raise:`.

Try running this test. Note that the first argument of the `should:` and `shouldnt:` methods is a block that contains the expression to be executed.

Running a single test

Normally, you will run your tests using the Test Runner or using your code browser. If you don't want to launch the Test Runner UI from the World menu, you can execute `TestRunner open`. You can also run a single test as follows:

```
MyExampleSetTest run: #testRemove
>>> 1 run, 1 passed, 0 failed, 0 errors
```

Running all the tests in a test class

Any subclass of `TestCase` responds to the message `suite`, which will build a test suite that contains all the methods in the class whose names start with the string *test*.

To run the tests in the suite, send it the message `run`. For example:

```
MyExampleSetTest suite run
>>> 4 run, 4 passed, 0 failed, 0 errors
```

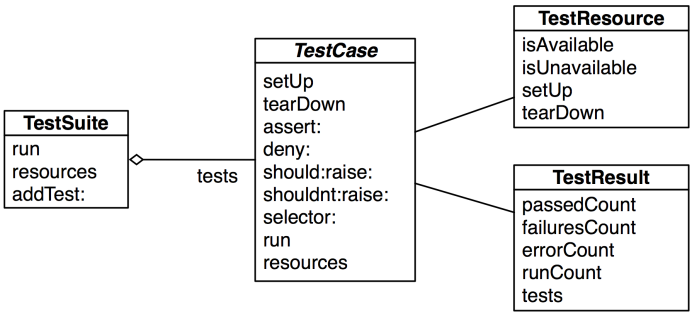

Figure 11-5 The four classes representing the core of SUnit.

Must I subclass TestCase?

In JUnit you can build a TestSuite from an arbitrary class containing `test*` methods. In SUnit you can do the same but you will then have to create a suite by hand and your class will have to implement all the essential `TestCase` methods like `assert:`. We recommend, however, that you not try to do this. The framework is there: use it.

11.10 The SUnit framework

SUnit consists of four main classes: `TestCase`, `TestSuite`, `TestResult`, and `TestResource`, as shown in Figure 11-5. The notion of a *test resource* represents a resource that is expensive to set-up but which can be used by a whole series of tests. A `TestResource` specifies a `setUp` method that is executed just once before a suite of tests; this is in distinction to the `TestCase >> setUp` method, which is executed before each test.

TestCase

`TestCase` is an abstract class that is designed to be subclassed. Each of its subclasses represents a group of tests that share a common context (that is, a test suite). Each test is run by creating a new instance of a subclass of `TestCase`, running `setUp`, running the test method itself, and then sending the `tearDown`.

The context is specified by instance variables of the subclass and by the specialization of the method `setUp`, which initializes those instance variables. Subclasses of `TestCase` can also override method `tearDown`, which is invoked after the execution of each test, and can be used to release any objects allocated during `setUp`.

TestSuite

Instances of the class `TestSuite` contain a collection of test cases. An instance of `TestSuite` contains tests, and other test suites. That is, a test suite contains sub-instances of `TestCase` and `TestSuite`.

Both individual test cases and test suites understand the same protocol, so they can be treated in the same way (for example, both can be `run`). This is in fact an application of the Composite pattern in which `TestSuite` is the composite and the test cases are the leaves.

TestResult

The class `TestResult` represents the results of a `TestSuite` execution. It records the number of tests passed, the number of tests failed, and the number of errors signalled.

TestResource

One of the important features of a suite of tests is that they should be independent of each other. The failure of one test should not cause an avalanche of failures of other tests that depend upon it, nor should the order in which the tests are run matter. Performing `setUp` before each test and `tearDown` afterwards helps to reinforce this independence.

However, there are occasions where setting up the necessary context is just too time-consuming for it to be done before the execution of each test. Moreover, if it is known that the test cases do not disrupt the resources used by the tests, then it is wasteful to set them up afresh for each test. It is sufficient to set them up once for each suite of tests. Suppose, for example, that a suite of tests needs to query a database, or do analysis on some compiled code. In such cases, it may make sense to set up the database and open a connection to it, or to compile some source code, before any of the tests start to run.

Where should we cache these resources, so that they can be shared by a suite of tests? The instance variables of a particular `TestCase` subclass won't do, because a TestCase instance persists only for the duration of a single test (as mentioned before, the instance is created anew *for each test method*). A global variable would work, but using too many global variables pollutes the name space, and the binding between the global and the tests that depend on it will not be explicit. A better solution is to put the necessary resources in a singleton object of some class. The class `TestResource` exists to be subclassed by such resource classes. Each subclass of `TestResource` understands the message `current`, which will answer a singleton instance of that subclass. Methods `setUp` and `tearDown` should be overridden in the subclass to ensure that the resource is initialized and finalized.

Listing 11-6 An example of a TestResource subclass

```
TestResource subclass: #MyTestResource
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'MyTestExample'

TestCase subclass: #MyTestCase
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'MyTestExample'

MyTestCase class >> resources
  "Associate the resource with this class of test cases"

  ^ { MyTestResource }
```

One thing remains: somehow, SUnit has to be told which resources are associated with which test suite. A resource is associated with a particular subclass of `TestCase` by overriding the *class* method `resources`.

By default, the resources of a `TestSuite` are the union of the resources of the `TestCases` that it contains.

Here is an example. We define a subclass of `TestResource` called `MyTestResource`. Then we associate it with `MyTestCase` by overriding the class method `MyTestCase class >> resources` to return an array of the test resource classes that `MyTestCase` will use.

Exercise

The following trace (written to the `Transcript`) illustrates that a global set up is run before and after each test in a sequence. Let's see if you can obtain this trace yourself.

```
MyTestResource >> setUp has run.
MyTestCase >> setUp has run.
MyTestCase >> testOne has run.
MyTestCase >> tearDown has run.
MyTestCase >> setUp has run.
MyTestCase >> testTwo has run.
MyTestCase >> tearDown has run.
MyTestResource >> tearDown has run.
```

Create new classes `MyTestResource` and `MyTestCase` which are subclasses of `TestResource` and `TestCase` respectively. Add the appropriate methods so that the following messages are written to the `Transcript` when you run your tests.

Solution.

You will need to write the following six methods.

```
MyTestCase >> setUp
  Transcript show: 'MyTestCase>>setUp has run.'; cr

MyTestCase >> tearDown
  Transcript show: 'MyTestCase>>tearDown has run.'; cr

MyTestCase >> testOne
  Transcript show: 'MyTestCase>>testOne has run.'; cr

MyTestCase >> testTwo
  Transcript show: 'MyTestCase>>testTwo has run.'; cr

MyTestResource >> setUp
  Transcript show: 'MyTestResource>>setUp has run'; cr

MyTestResource >> tearDown
  Transcript show: 'MyTestResource>>tearDown has run.'; cr
```

11.11 Chapter summary

This chapter explained why tests are an important investment in the future of your code. We explained in a step-by-step fashion how to define a few tests for the class `Set`. Then we gave an overview of the core of the SUnit framework by presenting the classes `TestCase`, `TestResult`, `TestSuite` and `TestResources`. Finally we looked deep inside SUnit by following the execution of a test and a test suite.

- To maximize their potential, unit tests should be fast, repeatable, independent of any direct human interaction and cover a single unit of functionality.
- Tests for a class called `MyClass` belong in a class named `MyClassTest`, which should be introduced as a subclass of `TestCase`.
- Initialize your test data in a `setUp` method.
- Each test method should start with the word *test*.
- Use the `TestCase` methods `assert:`, `deny:` and others to make assertions.
- Run tests!

CHAPTER 12

Basic classes

Pharo is a really simple language but powerful language. Part of its power is not in the language but in its class libraries. To program effectively in it, you will need to learn how the class libraries support the language and environment. The class libraries are entirely written in Pharo, and can easily be extended. (Recall that a package may add new functionality to a class even if it does not define this class.)

Our goal here is not to present in tedious detail the whole of the Pharo class library, but rather to point out the key classes and methods that you will need to use (or subclass/override) to program effectively. In this chapter, we will cover the basic classes that you will need for nearly every application: `Object`, `Number` and its subclasses, `Character`, `String`, `Symbol`, and `Boolean`.

12.1 Object

For all intents and purposes, `Object` is the root of the inheritance hierarchy. Actually, in Pharo the true root of the hierarchy is `ProtoObject`, which is used to define minimal entities that masquerade as objects, but we can ignore this point for the time being.

`Object` defines almost 400 methods (in other words, every class that you define will automatically provide all those methods). *Note:* You can count the number of methods in a class like so:

```
Object selectors size         "Count the instance methods in Object"
Object class selectors size   "Count the class methods"
```

Class `Object` provides default behaviour common to all normal objects, such as access, copying, comparison, error handling, message sending, and re-

Listing 12-1 printOn: redefinition.

```
Color >> printOn: aStream
  | name |
  (name := self name).
  name = #unnamed
    ifFalse: [
      ^ aStream
        nextPutAll: 'Color ';
        nextPutAll: name ].
  self storeOn: aStream
```

flection. Also utility messages that all objects should respond to are defined here. `Object` has no instance variables, nor should any be added. This is due to several classes of objects that inherit from `Object` that have special implementations (`SmallInteger` and `UndefinedObject` for example) that the VM knows about and depends on the structure and layout of certain standard classes.

If we begin to browse the method protocols on the instance side of `Object` we will start to see some of the key behaviour it provides.

12.2 Object printing

Every object can return a printed form of itself. You can select any expression in a textpane and select the `Print it` menu item: this executes the expression and asks the returned object to print itself. In fact this sends the message `printString` to the returned object. The method `printString`, which is a template method, at its core sends the message `printOn:` to its receiver. The message `printOn:` is a hook that can be specialized.

Method `Object>>printOn:` is very likely one of the methods that you will most frequently override. This method takes as its argument a `Stream` on which a `String` representation of the object will be written. The default implementation simply writes the class name preceded by `a` or `an`. `Object>>printString` returns the `String` that is written.

For example, the class `OpalCompiler` does not redefine the method `printOn:` and sending the message `printString` to an instance executes the methods defined in `Object`.

```
OpalCompiler new printString
>>> 'an OpalCompiler'
```

The class `Color` shows an example of `printOn:` specialization. It prints the name of the class followed by the name of the class method used to generate that color.

```
Color red printString
>>> 'Color red'
```

Note that the message `printOn:` is not the same as `storeOn:`. The message `storeOn:` writes to its argument stream an expression that can be used to recreate the receiver. This expression is executed when the stream is read using the message `readFrom:`. On the other hand, the message `printOn:` just returns a textual version of the receiver. Of course, it may happen that this textual representation may represent the receiver as a self-evaluating expression.

12.3 A word about representation and self-evaluating representation.

In functional programming, expressions return values when executed. In Pharo, message sends (expressions) return objects (values). Some objects have the nice property that their value is themselves. For example, the value of the object `true` is itself i.e., the object `true`. We call such objects *self-evaluating objects*. You can see a *printed* version of an object value when you print the object in a playground. Here are some examples of such self-evaluating expressions.

```
true
>>> true
```

```
3@4
>>> (3@4)
```

```
$a
>>> $a
```

```
#(1 2 3)
>>> #(1 2 3)
```

```
Color red
>>> Color red
```

Note that some objects such as arrays are self-evaluating or not depending on the objects they contain. For example, an array of booleans is self-evaluating, whereas an array of persons is not. The following example shows that a dynamic array is self-evaluating only if its elements are:

```
{10@10. 100@100}
>>> {(10@10). (100@100)}
```

```
{OpalCompiler new . 100@100}
>>> an Array(an OpalCompiler (100@100))
```

Remember that literal arrays can only contain literals. Hence the following array does not contain two points but rather six literal elements.

```
#(10@10 100@100)
>>> #(10 #@ 10 100 #@ 100)
```

Listing 12-2 Self-evaluation of `Point`

```
Point >> printOn: aStream
  "The receiver prints on aStream in terms of infix notation."

  aStream nextPut: $(.
  x printOn: aStream.
  aStream nextPut: $@.
  (y notNil and: [y negative])
    ifTrue: [
      "Avoid ambiguous @- construct"
      aStream space ].
  y printOn: aStream.
  aStream nextPut: $).
```

Listing 12-3 Self-evaluation of `Interval`

```
Interval >> printOn: aStream
    aStream nextPut: $(;
        print: start;
        nextPutAll: ' to: ';
        print: stop.
    step ~= 1 ifTrue: [aStream nextPutAll: ' by: '; print: step].
    aStream nextPut: $)
```

Listing 12-4 Object equality

```
Object >> = anObject
    "Answer whether the receiver and the argument represent the same
    object.
    If = is redefined in any subclass, consider also redefining the
    message hash."

    ^ self == anObject
```

Lots of `printOn:` method specializations implement self-evaluating behavior. The implementations of `Point>>printOn:` and `Interval>>printOn:` are self-evaluating.

```
1 to: 10
>>> (1 to: 10)    "intervals are self-evaluating"
```

12.4 Identity and equality

In Pharo, the message `=` tests object *equality* while the message `==` tests object *identity*. The former is used to check whether two objects represent the same value, while the latter is used to check whether two expressions represent the same object.

The default implementation of object equality is to test for object identity:

If you override =, you should consider overriding `hash`. If instances of your class are ever used as keys in a `Dictionary`, then you should make sure that instances that are considered to be equal have the same hash value.

Although you should override = and `hash` together, you should *never* override ==. The semantics of object identity is the same for all classes. Message == is a primitive method of `ProtoObject`.

Note that Pharo has some strange equality behaviour compared to other Smalltalks. For example a symbol and a string can be equal. (We consider this to be a bug, not a feature.)

```
#'lulu' = 'lulu'
>>> true
```

```
'lulu' = #'lulu'
>>> true
```

12.5 Class membership

Several methods allow you to query the class of an object.

`class`.

You can ask any object about its class using the message `class`.

```
1 class
>>> SmallInteger
```

`isMemberOf:`.

Conversely, you can ask if an object is an instance of a specific class:

```
1 isMemberOf: SmallInteger
>>> true    "must be precisely this class"
```

```
1 isMemberOf: Integer
>>> false
```

```
1 isMemberOf: Number
>>> false
```

```
1 isMemberOf: Object
>>> false
```

Since Pharo is written in itself, you can really navigate through its structure using the right combination of superclass and class messages (see Chapter: Classes and Metaclasses).

`isKindOf:`.

`Object>>isKindOf:` answers whether the receiver's class is either the same as, or a subclass of the argument class.

```
1 isKindOf: SmallInteger
>>> true
```

```
1 isKindOf: Integer
>>> true
```

```
1 isKindOf: Number
>>> true
```

```
1 isKindOf: Object
>>> true
```

```
1 isKindOf: String
>>> false
```

```
1/3 isKindOf: Number
>>> true
```

```
1/3 isKindOf: Integer
>>> false
```

`1/3` which is a `Fraction` is a kind of `Number`, since the class `Number` is a superclass of the class `Fraction`, but `1/3` is not an `Integer`.

`respondsTo:`.

`Object>>respondsTo:` answers whether the receiver understands the message selector given as an argument.

```
1 respondsTo: #,
>>> false
```

A note on the usage of `respondsTo:`. Normally it is a bad idea to query an object for its class, or to ask it which messages it understands. Instead of making decisions based on the class of object, you should simply send a message to the object and let it decide (on the basis of its class) how it should behave. This concept is sometimes referred to as *duck typing*.

12.6 Copying

Copying objects introduces some subtle issues. Since instance variables are accessed by reference, a *shallow copy* of an object would share its references to instance variables with the original object:

```
a1 := { { 'harry' } }.
a1
>>> #(#('harry'))
```

Listing 12-5 Copying objects as a template method

```
Object >> copy
    "Answer another instance just like the receiver.
    Subclasses typically override postCopy;
    they typically do not override shallowCopy."

    ^ self shallowCopy postCopy
```

```
a2 := a1 shallowCopy.
a2
>>> #(#('harry'))
```

```
(a1 at: 1) at: 1 put: 'sally'.
a1
>>> #(#('sally'))
```

```
a2
>>> #(#('sally'))    "the subarray is shared!"
```

`Object>>shallowCopy` is a primitive method that creates a shallow copy of an object. Since `a2` is only a shallow copy of `a1`, the two arrays share a reference to the nested `Array` that they contain.

`Object>>deepCopy` makes an arbitrarily deep copy of an object.

```
a1 := { { { 'harry' } } } .
a2 := a1 deepCopy.
(a1 at: 1) at: 1 put: 'sally'.
a1
>>> #(#('sally'))
```

```
a2
>>> #(#(#('harry')))
```

The problem with `deepCopy` is that it will not terminate when applied to a mutually recursive structure:

```
a1 := { 'harry' }.
a2 := { a1 }.
a1 at: 1 put: a2.
a1 deepCopy
>>> !''... does not terminate!''!
```

An alternate solution is to use message `copy`. It is implemented on `Object` as follows:

```
Object >> postCopy
    ^ self
```

By default `postCopy` returns self. It means that by default `copy` is doing the same as `shallowCopy` but each subclass can decide to customise the `postCopy` method which acts as a hook. You should override `postCopy` to copy

Listing 12-6 Checking a pre-condition

```
Stack >> pop
    "Return the first element and remove it from the stack."

    self assert: [ self isNotEmpty ].
    ^ self linkedList removeFirst element
```

any instance variables that should not be shared. In addition there is a good chance that `postCopy` should always do a `super postCopy` to ensure that state of the superclass is also copied.

12.7 Debugging

`halt`.

The most important method here is `halt`. To set a breakpoint in a method, simply insert the expression `self halt` at some point in the body of the method. (Note that since `halt` is defined on `Object` you can also write `1 halt`). When this message is sent, execution will be interrupted and a debugger will open to this point in your program (see Chapter : The Pharo Environment for more details about the debugger).

You can also use `Halt once` or `Halt if: aCondition`. Have a look at the class `Halt` which is an special exception.

`assert:`.

The next most important message is `assert:`, which expects a block as its argument. If the block evaluates to `true`, execution continues. Otherwise an `AssertionFailure` exception will be raised. If this exception is not otherwise caught, the debugger will open to this point in the execution. `assert:` is especially useful to support *design by contract*. The most typical usage is to check non-trivial pre-conditions to public methods of objects. `Stack>>pop` could easily have been implemented as follows (note that this definition is anhypothetical example and not in the Pharo 8.0 system):

Do not confuse `Object>>assert:` with `TestCase>>assert:`, which occurs in the SUnit testing framework (see Chapter : SUnit). While the former expects a block as its argument (actually, it will take any argument that understands `value`, including a `Boolean`), the latter expects a `Boolean`. Although both are useful for debugging, they each serve a very different purpose.

12.8 Error handling

This protocol contains several methods useful for signaling run-time errors.

Listing 12-7 Signaling that a method is abstract

```
Object >> subclassResponsibility
    "This message sets up a framework for the behavior of the class'
    subclasses.
    Announce that the subclass should have implemented this message."

    SubclassResponsibility signalFor: thisContext sender selector
```

`deprecated:`.

Sending `self deprecated:` signals that the current method should no longer be used, if deprecation has been turned on. You can turn it on/off in the Debugging section using the Settings browser. The argument should describe an alternative. Look for senders of the message `deprecated:` to get an idea.

`doesNotUnderstand:`.

`doesNotUnderstand:` (commonly abbreviated in discussions as DNU or MNU) is sent whenever message lookup fails. The default implementation, i.e., `Object>>doesNotUnderstand:` will trigger the debugger at this point. It may be useful to override `doesNotUnderstand:` to provide some other behaviour.

`error`.

`Object>>error` and `Object>>error:` are generic methods that can be used to raise exceptions. (Generally it is better to raise your own custom exceptions, so you can distinguish errors arising from your code from those coming from kernel classes.)

`subclassResponsibility`.

Abstract methods are implemented by convention with the body `self subclassResponsibility`. Should an abstract class be instantiated by accident, then calls to abstract methods will result in `Object>>subclassResponsibility` being executed.

`Magnitude`, `Number`, and `Boolean` are classical examples of abstract classes that we shall see shortly in this chapter.

```
Number new + 1
>>> !''Error: Number is an abstract class. Make a concrete
    subclass.''!
```

Listing 12-8 `initialize` as an empty hook method

```
ProtoObject >> initialize
   "Subclasses should redefine this method to perform
    initializations on instance creation"
```

`shouldNotImplement`.

`self shouldNotImplement` is sent by convention to signal that an inherited method is not appropriate for this subclass. This is generally a sign that something is not quite right with the design of the class hierarchy. Due to the limitations of single inheritance, however, sometimes it is very hard to avoid such workarounds.

A typical example is `Collection>>remove:` which is inherited by `Dictionary` but flagged as not implemented. (A `Dictionary` provides `removeKey:` instead.)

12.9 Testing

The `testing` methods have nothing to do with SUnit testing! A testing method is one that lets you ask a question about the state of the receiver and returns a `Boolean`.

Numerous testing methods are provided by `Object`. There are `isArray`, `isBoolean`, `isBlock`, `isCollection` and so on. Generally such methods are to be avoided since querying an object for its class is a form of violation of encapsulation. Instead of testing an object for its class, one should simply send a request and let the object decide how to handle it.

Nevertheless some of these testing methods are undeniably useful. The most useful are probably `ProtoObject>>isNil` and `Object>>notNil` (though the Null Object design pattern can obviate the need for even these methods).

12.10 Initialize

A final key method that occurs not in `Object` but in `ProtoObject` is `initialize`.

The reason this is important is that in Pharo, the default `new` method defined for every class in the system will send `initialize` to newly created instances.

This means that simply by overriding the `initialize` hook method, new instances of your class will automatically be initialized. The `initialize` method should normally perform a `super initialize` to establish the class invariant for any inherited instance variables.

Listing 12-9 new as a class-side template method

```
Behavior >> new
    "Answer a new initialized instance of the receiver (which is a
    class) with no indexable
    variables. Fail if the class is indexable."
    ^ self basicNew initialize
```

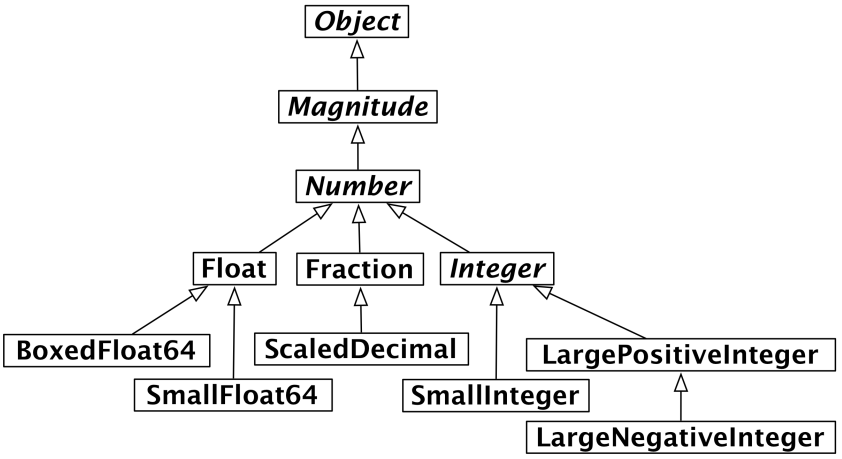

Figure 12-10 The number hierarchy.

12.11 Numbers

Numbers in Pharo are not primitive data values but true objects. Of course numbers are implemented efficiently in the virtual machine, but the `Number` hierarchy is as perfectly accessible and extensible as any other portion of the class hierarchy.

The abstract root of this hierarchy is `Magnitude`, which represents all kinds of classes supporting comparison operators. `Number` adds various arithmetic and other operators as mostly abstract methods. `Float` and `Fraction` represent, respectively, floating point numbers and fractional values. `Float` subclasses (`BoxedFloat64` and `SmallFloat64`) represent Float on certain architectures. For example `BoxedFloat64` is only available for 64 bit systems. `Integer` is also abstract, thus distinguishing between subclasses `SmallInteger`, `LargePositiveInteger` and `LargeNegativeInteger`. For the most part, users do not need to be aware of the difference between the three `Integer` classes, as values are automatically converted as needed.

Listing 12-11 Abstract comparison methods

```
Magnitude >> < aMagnitude
    "Answer whether the receiver is less than the argument."

    ^ self subclassResponsibility

Magnitude >> > aMagnitude
    "Answer whether the receiver is greater than the argument."

    ^ aMagnitude < self
```

12.12 Magnitude

`Magnitude` is the parent not only of the `Number` classes, but also of other classes supporting comparison operations, such as `Character`, `Duration` and `Timespan`.

Methods < and = are abstract. The remaining operators are generically defined. For example:

12.13 Number

Similarly, `Number` defines `+`, `-`, `*` and `/` to be abstract, but all other arithmetic operators are generically defined.

All `Number` objects support various *converting* operators, such as `asFloat` and `asInteger`. There are also numerous *shortcut constructor methods* which generate `Durations`, such as `hour`, `day` and `week`.

`Numbers` directly support common *math functions* such as `sin`, `log`, `raiseTo:`, `squared`, `sqrt` and so on.

The method `Number>>printOn:` is implemented in terms of the abstract method `Number>>printOn:base:`. (The default base is 10.)

Testing methods include `even`, `odd`, `positive` and `negative`. Unsurprisingly `Number` overrides `isNumber`. More interestingly, `isInfinite` is defined to return `false`.

Truncation methods include `floor`, `ceiling`, `integerPart`, `fractionPart` and so on.

```
1 + 2.5
>>> 3.5             "Addition of two numbers"
```

```
3.4 * 5
>>> 17.0           "Multiplication of two numbers"
```

```
8 / 2
>>> 4                 "Division of two numbers"
```

```
10 - 8.3
>>> 1.7              "Subtraction of two numbers"
```

```
12 = 11
>>> false           "Equality between two numbers"
```

```
12 ~= 11
>>> true            "Test if two numbers are different"
```

```
12 > 9
>>> true            "Greater than"
```

```
12 >= 10
>>> true            "Greater or equal  than"
```

```
12 < 10
>>> false           "Smaller than"
```

```
100@10
>>> 100@10    "Point creation"
```

The following example works surprisingly well in Pharo:

```
1000 factorial / 999 factorial
>>> 1000
```

Note that `1000 factorial` is really calculated, which in many other languages can be quite difficult to compute. This is an excellent example of automatic coercion and exact handling of a number.

> **To do** Try to display the result of `1000 factorial`. It takes more time to display it than to calculate it!

12.14 Float

`Float` implements the abstract `Number` methods for floating point numbers.

More interestingly, `Float class` (i.e., the class-side of `Float`) provides methods to return the following *constants*: `e`, `infinity`, `nan` and `pi`.

```
Float pi
>>> 3.141592653589793
```

```
Float infinity
>>> Float infinity
```

```
Float infinity isInfinite
>>> true
```

12.15 Fraction

`Fractions` are represented by instance variables for the numerator and denominator, which should be `Integers`. `Fractions` are normally created by

`Integer` division (rather than using the constructor method `Fraction>>numerator:denominator:`):

```
6/8
>>> (3/4)
```

```
(6/8) class
>>> Fraction
```

Multiplying a `Fraction` by an `Integer` or another `Fraction` may yield an `Integer`:

```
6/8 * 4
>>> 3
```

12.16 Integer

`Integer` is the abstract parent of three concrete integer implementations. In addition to providing concrete implementations of many abstract `Number` methods, it also adds a few methods specific to integers, such as `factorial`, `atRandom`, `isPrime`, `gcd:` and many others.

`SmallInteger` is special in that its instances are represented compactly — instead of being stored as a reference, a `SmallInteger` is represented directly using the bits that would otherwise be used to hold a reference. The first bit of an object reference indicates whether the object is a `SmallInteger` or not. Now the virtual machine abstracts that from you, therefore you cannot see this directly when inspecting the object.

The class methods `minVal` and `maxVal` tell us the range of a `SmallInteger`, note that it varies depending on the size of your image from either `(2 raisedTo: 30) - 1` for a 32-bits image or `(2 raisedTo: 60) - 1` for a 64-bits one.

```
SmallInteger maxVal = ((2 raisedTo: 30) - 1)
>>> true
```

```
SmallInteger minVal = (2 raisedTo: 30) negated
>>> true
```

When a `SmallInteger` goes out of this range, it is automatically converted to a `LargePositiveInteger` or a `LargeNegativeInteger`, as needed:

```
(SmallInteger maxVal + 1) class
>>> LargePositiveInteger
```

```
(SmallInteger minVal - 1) class
>>> LargeNegativeInteger
```

Large integers are similarly converted back to small integers when appropriate.

As in most programming languages, integers can be useful for specifying iterative behaviour. There is a dedicated method `timesRepeat:` for evaluating a block repeatedly. We have already seen a similar example in Chapter : Syntax in a Nutshell.

```
| n |
n := 2.
3 timesRepeat: [ n := n * n ].
n
>>> 256
```

12.17 Characters

`Character` is defined a subclass of `Magnitude`. Printable characters are represented in Pharo as `$<char>`. For example:

```
$a < $b
>>> true
```

Non-printing characters can be generated by various class methods. `Character class>>value:` takes the Unicode (or ASCII) integer value as argument and returns the corresponding character. The protocol `accessing untypeable characters` contains a number of convenience constructor methods such as `backspace`, `cr`, `escape`, `euro`, `space`, `tab`, and so on.

```
Character space = (Character value: Character space asciiValue)
>>> true
```

The `printOn:` method is clever enough to know which of the three ways to generate characters offers the most appropriate representation:

```
Character value: 1
>>> Character home
```

```
Character value: 2
>>> Character value: 2
```

```
Character value: 32
>>> Character space
```

```
Character value: 97
>>> $a
```

Various convenient *testing* methods are built in: `isAlphaNumeric`, `isCharacter`, `isDigit`, `isLowercase`, `isVowel`, and so on.

To convert a `Character` to the string containing just that character, send `asString`. In this case `asString` and `printString` yield different results:

```
$a asString
>>> 'a'
```

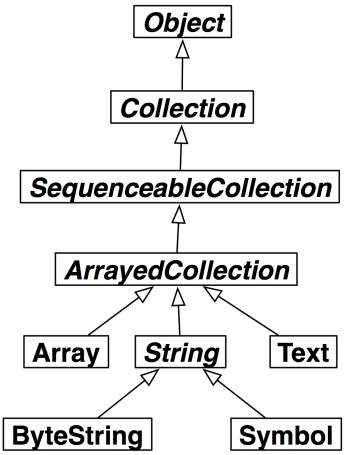

Figure 12-12 The String Hierarchy.

```
$a
>>> $a
```

```
$a printString
>>> '$a'
```

Like `SmallInteger`, a `Character` is a immediate value not a object reference. Most of the time you won't see any difference and can use objects of class `Character` like any other too. But this means, equal value characters are always *identical*:

```
(Character value: 97) == $a
>>> true
```

12.18 Strings

A `String` is an indexed `Collection` that holds only `Characters`.

In fact, `String` is abstract and Pharo strings are actually instances of the concrete class `ByteString`.

```
'hello world' class
>>> ByteString
```

The other important subclass of `String` is `Symbol`. The key difference is that there is only ever a single instance of `Symbol` with a given value. (This is sometimes called *the unique instance property*). In contrast, two separately constructed `Strings` that happen to contain the same sequence of characters will often be different objects.

```
'hel','lo' == 'hello'
>>> false
```

```
('hel','lo') asSymbol == #hello
>>> true
```

Another important difference is that a `String` is mutable, whereas a `Symbol` is immutable.

```
'hello' at: 2 put: $u; yourself
>>> 'hullo'
```

```
#hello at: 2 put: $u
>>> Error: symbols can not be modified.
```

It is easy to forget that since strings are collections, they understand the same messages that other collections do:

```
#hello indexOf: $o
>>> 5
```

Although `String` does not inherit from `Magnitude`, it does support the usual `comparing` methods, `<`, `=` and so on. In addition, `String>>match:` is useful for some basic glob-style pattern-matching:

```
'*or*' match: 'zorro'
>>> true
```

Regular expressions will be discussed in more detail in Chapter : Regular Expressions in Pharo.

Strings support a rather large number of conversion methods. Many of these are shortcut constructor methods for other classes, such as `asDate`, `asInteger` and so on. There are also a number of useful methods for converting a string to another string, such as `capitalized` and `translateToLowercase`.

For more on strings and collections, see Chapter : Collections.

12.19 Booleans

The class `Boolean` offers a fascinating insight into how much of the Pharo language has been pushed into the class library. `Boolean` is the abstract superclass of the singleton classes `True` and `False`.

Most of the behaviour of `Booleans` can be understood by considering the method `ifTrue:ifFalse:`, which takes two `Blocks` as arguments.

```
4 factorial > 20
  ifTrue: [ 'bigger' ]
  ifFalse: [ 'smaller' ]
>>> 'bigger'
```

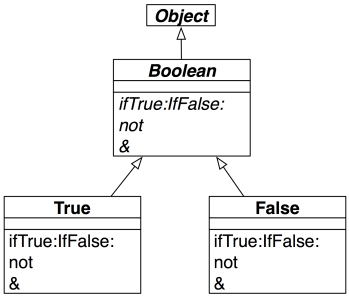

Figure 12-13 The Boolean Hierarchy.

Listing 12-14 Implementations of `ifTrue:ifFalse:`

```
True >> ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ^ trueAlternativeBlock value

False >> ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ^ falseAlternativeBlock value
```

Listing 12-15 Implementing negation

```
True >> not
    "Negation--answer false since the receiver is true."
    ^ false

False >> not
    "Negation--answer true since the receiver is false."
    ^ true
```

The method `ifTrue:ifFalse:` is abstract in class `Boolean`. The implementations in its concrete subclasses are both trivial:

Each of them execute the correct block depending on the receiver of the message. In fact, this is the essence of OOP: when a message is sent to an object, the object itself determines which method will be used to respond. In this case an instance of `True` simply executes the *true* alternative, while an instance of `False` executes the *false* alternative. All the abstract `Boolean` methods are implemented in this way for `True` and `False`. For example the implementation of negation (message `not`) is defined the same way:

`Boolean`s offer several useful convenience methods, such as `ifTrue:`, `ifFalse:`, and `ifFalse:ifTrue`. You also have the choice between eager and lazy conjunctions and disjunctions.

```
( 1 > 2 ) & ( 3 < 4 )
>>> false    "Eager, must evaluate both sides"
```

```
( 1 > 2 ) and: [ 3 < 4 ]
>>> false    "Lazy, only evaluate receiver"
```

```
( 1 > 2 ) and: [ ( 1 / 0 ) > 0 ]
>>> false    "argument block is never executed, so no exception"
```

In the first example, both `Boolean` subexpressions are executed, since `&` takes a `Boolean` argument. In the second and third examples, only the first is executed, since `and:` expects a `Block` as its argument. The `Block` is executed only if the first argument is `true`.

Try to imagine how `and:` and `or:` are implemented. Check the implementations in `Boolean`, `True` and `False`.

12.20 Chapter summary

- If you override = then you should override `hash` as well.
- Override `postCopy` to correctly implement copying for your objects.
- Use `self halt.` to set a breakpoint.
- Return `self subclassResponsibility` to make a method abstract.
- To give an object a `String` representation you should override `printOn:`.
- Override the hook method `initialize` to properly initialize instances.
- `Number` methods automatically convert between `Floats`, `Fractions` and `Integers`.
- `Fractions` truly represent rational numbers rather than floats.
- All `Characters` are like unique instances.
- `Strings` are mutable; `Symbols` are not. Take care not to mutate string literals, however!
- `Symbols` are unique; `Strings` are not.
- `Strings` and `Symbols` are `Collections` and therefore support the usual `Collection` methods.

CHAPTER 13

Collections

13.1 Introduction

The collection classes form a loosely-defined group of general-purpose subclasses of `Collection` and `Stream`. Many of these (like `Bitmap`, `FileStream` and `CompiledMethod`) are special-purpose classes crafted for use in other parts of the system or in applications, and hence not categorized as *Collections* by the system organization. For the purposes of this chapter, we use the term *Collection Hierarchy* to mean `Collection` and its subclasses that are *also* in the packages labelled `Collections-*`. We use the term *Stream Hierarchy* to mean `Stream` and its subclasses that are *also* in the `Collections-Streams` packages.

In this chapter we focus mainly on the subset of collection classes shown in Figure 13-1. Streams will be discussed separately in Chapter : Streams.

13.2 The varieties of collections

To make good use of the collection classes, the reader needs at least a superficial knowledge of the wide variety of collections that they implement, and their commonalities and differences.

Programming with collections using high-order functions rather than individual elements is an important way to raise the level of abstraction of a program. The Lisp function `map`, which applies an argument function to every element of a list and returns a new list containing the results is an early example of this style. Following its Smalltalk root, Pharo adopts this collection-based high-order programming as a central tenet. Modern functional programming languages such as ML and Haskell have followed Smalltalk's lead.

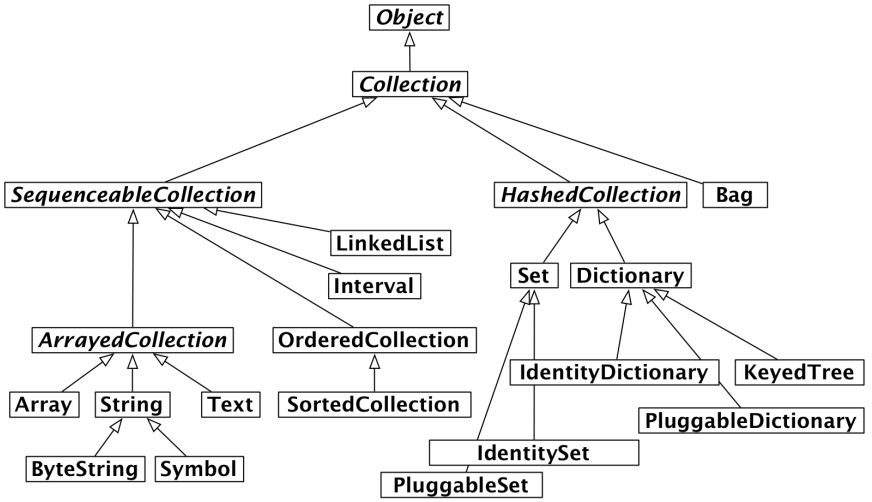

Figure 13-1 Some of the key collection classes in Pharo.

Why is this a good idea? Let us suppose you have a data structure containing a collection of student records, and wish to perform some action on all of the students that meet some criteria. Programmers raised to use an imperative language will immediately reach for a loop, but the Pharo programmer will write:

```
students
  select: [ :each | each gpa < threshold ]
```

This expression returns a new collection containing precisely those elements of `students` for which the block (the bracketed function) returns `true`. The block can be thought of as a lambda-expression defining an anonymous function `x. x gpa < threshold`. This code has the simplicity and elegance of a domain-specific query language.

The message `select:` is understood by *all* collections in Pharo. There is no need to find out if the student data structure is an array or a linked list: the `select:` message is understood by both. Note that this is quite different from using a loop, where one must know whether `students` is an array or a linked list before the loop can be set up.

In Pharo, when one speaks of a collection without being more specific about the kind of collection, one means an object that supports well-defined protocols for testing membership and enumerating the elements. *All* collections understand the `testing` messages `includes:`, `isEmpty` and `occurrencesOf:`. *All* collections understand the `enumeration` messages `do:`, `select:`, `reject:` (which is the opposite of `select:`), `collect:` (which is like Lisp's `map`), `detect:ifNone:`, `inject:into:` (which performs a left fold) and many

more. It is the ubiquity of this protocol, as well as its variety, that makes it so powerful.

The table below summarizes the standard protocols supported by most of the classes in the collection hierarchy. These methods are defined, redefined, optimized or occasionally even forbidden by subclasses of `Collection`.

Protocol	Methods
`accessing`	`size`, `capacity`, `at:`, `at:put:`
`testing`	`isEmpty`, `includes:`, `contains:`, `occurrencesOf:`
`adding`	`add:`, `addAll:`
`removing`	`remove:`, `remove:ifAbsent:`, `removeAll:`
`enumerating`	`do:`, `collect:`, `select:`, `reject:` `detect:`, `detect:ifNone:`, `inject:into:`
`converting`	`asBag`, `asSet`, `asOrderedCollection`, `asSortedCollection`, `asArray`, `asSortedCollection:`
`creation`	`with:`, `with:with:`, `with:with:with:`, `with:with:with:with:`, `withAll:`

Beyond this basic uniformity, there are many different kinds of collections either supporting different protocols or providing different behaviour for the same requests. Let us briefly observe some of the key differences:

`Sequenceable:` Instances of all subclasses of `SequenceableCollection` start from a `first` element and proceed in a well-defined order to a `last` element. Instances of `Set`, `Bag` and `Dictionary`, on the other hand, are not sequenceable.

`Sortable:` A `SortedCollection` maintains its elements in sort order.

`Indexable:` Most sequenceable collections are also indexable, that is, elements can be retrieved with message `at: anIndex`. `Array` is the familiar indexable data structure with a fixed size; `anArray at: n` retrieves the n^{th} element of `anArray`, and `anArray at: n put: v` changes the n^{th} element to `v`. `LinkedLists` and `SkipLists` are sequenceable but not indexable, that is, they understand `first` and `last`, but not the message `at:`.

`Keyed:` Instances of `Dictionary` and its subclasses are accessed by keys instead of indices.

`Mutable:` Most collections are mutable, but `Intervals` and `Symbols` are not. An `Interval` is an immutable collection representing a range of `Integers`. For example, `5 to: 16 by: 2` is an interval that contains the elements 5, 7, 9, 11, 13 and 15. It is indexable with message `at: anIndex`, but cannot be changed with message `at: anIndex put: aValue`.

`Growable:` Instances of `Interval` and `Array` are always of a fixed size. Other kinds of collections (sorted collections, ordered collections, and

Arrayed Implementation	Ordered Implementation	Hashed Implementation	Linked Implementation	Interval Implementation
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

Figure 13-2 Some collection classes categorized by implementation technique.

linked lists) can grow after creation. The class `OrderedCollection` is more general than `Array`; the size of an `OrderedCollection` grows on demand, and it defines messages `addFirst: anElement` and `addLast: anElement` as well as messages `at: anIndex` and `at: anIndex put: aValue`.

`Accepts duplicates:` A `Set` filters out duplicates, but a `Bag` does not. Classes `Dictionary`, `Set` and `Bag` use the `=` method provided by the elements; the `Identity` variants of these classes use the `==` method, which tests whether the arguments are the same object, and the `Pluggable` variants use an arbitrary equivalence relation supplied by the creator of the collection.

`Heterogeneous:` Most collections will hold any kind of element. A `String`, `CharacterArray` or `Symbol`, however, only holds `Characters`. An `Array` will hold any mix of objects, but a `ByteArray` only holds Bytes. A `LinkedList` is constrained to hold elements that conform to the `Link accessing` protocol.

13.3 Collection implementations

These categorizations by functionality are not our only concern; we must also consider how the collection classes are implemented. As shown in Figure 13-2, five main implementation techniques are employed.

- Arrays store their elements in the (indexable) instance variables of the collection object itself; as a consequence, arrays must be of a fixed size, but can be created with a single memory allocation.
- `OrderedCollections` and `SortedCollections` store their elements in an array that is referenced by one of the instance variables of the collection. Consequently, the internal array can be replaced with a larger one if the collection grows beyond its storage capacity.
- The various kinds of set and dictionary also reference a subsidiary array for storage, but use the array as a hash table. Bags use a subsidiary

`Dictionary`, with the elements of the bag as keys and the number of occurrences as values.

- LinkedLists use a standard singly-linked representation.
- Intervals are represented by three integers that record the two endpoints and the step size.

In addition to these classes, there are also *weak* variants of `Array`, `Set` and of the various kinds of dictionary. These collections hold onto their elements weakly, i.e., in a way that does not prevent the elements from being garbage collected. The Pharo virtual machine is aware of these classes and handles them specially.

13.4 Examples of key classes

We present now the most common or important collection classes using simple code examples. The main protocols of collections are:

- messages `at:`, `at:put:` — to access an element,
- messages `add:`, `remove:` — to add or remove an element,
- messages `size`, `isEmpty`, `include:` — to get some information about the collection,
- messages `do:`, `collect:`, `select:` — to iterate over the collection.

Each collection may implement (or not) such protocols, and when they do, they interpret them to fit with their semantics. We suggest you to browse the classes themselves in order to identify specific and more advanced protocols.

We will focus on the most common collection classes: `OrderedCollection`, `Set`, `SortedCollection`, `Dictionary`, `Interval`, and `Array`.

13.5 Common creation protocol.

There are several ways to create instances of collections. The most generic ones use the message `new: aSize` and `with: anElement`.

- `new: anInteger` creates a collection of size `anInteger` whose elements will all be `nil`.
- `with: anObject` creates a collection and adds `anObject` to the created collection.

Different collections will realize this behaviour differently.

You can create collections with initial elements using the methods `with:`, `with:with:` etc. for up to six elements.

```
Array with: 1
>>> #(1)
```

```
Array with: 1 with: 2
>>> #(1 2)
```

```
Array with: 1 with: 2 with: 3
>>> #(1 2 3)
```

```
Array with: 1 with: 2 with: 3 with: 4
>>> #(1 2 3 4)
```

```
Array with: 1 with: 2 with: 3 with: 4 with: 5
>>> #(1 2 3 4 5)
```

```
Array with: 1 with: 2 with: 3 with: 4 with: 5 with: 6
>>> #(1 2 3 4 5 6)
```

You can also use message `addAll: aCollection` to add all elements of one kind of collection to another kind:

```
(1 to: 5) asOrderedCollection addAll: '678'; yourself
>>> an OrderedCollection(1 2 3 4 5 6 7 8)
```

Take care that `addAll:` returns its argument, and not the receiver!

You can also create many collections with `withAll: aCollection`.

```
Array withAll: #(7 3 1 3)
>>> #(7 3 1 3)
```

```
OrderedCollection withAll: #(7 3 1 3)
>>> an OrderedCollection(7 3 1 3)
```

```
SortedCollection withAll: #(7 3 1 3)
>>> a SortedCollection(1 3 3 7)
```

```
Set withAll: #(7 3 1 3)
>>> a Set(7 1 3)
```

```
Bag withAll: #(7 3 1 3)
>>> a Bag(7 1 3 3)
```

13.6 Array

An `Array` is a fixed-sized collection of elements accessed by integer indices. Contrary to the C convention in Pharo, the first element of an array is at position 1 and not 0. The main protocol to access array elements is the method `at:` and `at:put:`.

- `at: anInteger` returns the element at index `anInteger`.
- `at: anInteger put: anObject` puts `anObject` at index `anInteger`.

Arrays are fixed-size collections therefore we cannot add or remove elements at the end of an array. The following code creates an array of size 5, puts values in the first 3 locations and returns the first element.

```
| anArray |
anArray := Array new: 5.
anArray at: 1 put: 4.
anArray at: 2 put: 3/2.
anArray at: 3 put: 'ssss'.
anArray at: 1
>>> 4
```

There are several ways to create instances of the class `Array`. We can use

- `new:`, `with:`,
- `#( )` construct - literal arrays and
- `{ . }` dynamic compact syntax.

Creation with `new:`

The message `new: anInteger` creates an array of size `anInteger`. `Array new: 5` creates an array of size 5.

Note The value of each element is initialized to `nil`.

Creation using `with:`

The `with:*` messages allow one to specify the value of the elements. The following code creates an array of three elements consisting of the number 4, the fraction 3/2 and the string `'lulu'`.

```
Array with: 4 with: 3/2 with: 'lulu'
>>>  {4. (3/2). 'lulu'}
```

Literal creation with `#( )`

The expression `#( )` creates literal arrays with constants or *literal* elements that have to be known when the expression is compiled, and not when it is executed. The following code creates an array of size 2 where the first element is the (literal) number 1 and the second the (literal) string `'here'`.

```
#(1 'here') size
>>> 2
```

Now, if you execute the expression `#(1+2)`, you do not get an array with a single element 3 but instead you get the array `#(1 #+ 2)` i.e., with three elements: 1, the symbol `#+` and the number 2.

```
#(1+2)
>>>  #(1 #+ 2)
```

This occurs because the construct `#( )` does not execute the expressions it contains. The elements are only objects that are created when parsing the expression (called literal objects). The expression is scanned and the resulting elements are fed to a new array. Literal arrays contain numbers, `nil`, `true`, `false`, symbols, strings and other literal arrays. During the execution of `#( )` expressions, there are no messages sent.

Dynamic creation with { . }

Finally, you can create a dynamic array using the construct `{ . }`. The expression `{ a . b }` is totally equivalent to `Array with: a with: b`. This means in particular that the expressions enclosed by { and } are executed (contrary to the case of `#( )`).

```
{ 1 + 2 }
>>> #(3)
```

```
{(1/2) asFloat} at: 1
>>> 0.5
```

```
{10 atRandom. 1/3} at: 2
>>> (1/3)
```

Element Access

Elements of all sequenceable collections can be accessed with messages `at: anIndex` and `at: anIndex put: anObject`.

```
| anArray |
anArray := #(1 2 3 4 5 6) copy.
anArray at: 3 >>> 3
anArray at: 3 put: 33.
anArray at: 3
>>> 33
```

Be careful: general principle is that literal arrays are not be modified! Literal arrays are kept in compiled method literal frames (a space where literals appearing in a method are stored), therefore unless you copy the array, the second time you execute the code your *literal* array may not have the value you expect. In the example, without copying the array, the second time around, the literal `#(1 2 3 4 5 6)` will actually be `#(1 2 33 4 5 6)`! Dynamic arrays do not have this problem because they are not stored in literal frames.

13.7 OrderedCollection

OrderedCollection is one of the collections that can grow, and to which elements can be added sequentially. It offers a variety of messages such as add:, addFirst:, addLast:, and addAll:.

```
| ordCol |
ordCol := OrderedCollection new.
ordCol add: 'Seaside'; add: 'SmalltalkHub'; addFirst: 'GitHub'.
ordCol
>>> an OrderedCollection('GitHub' 'Seaside' 'SmalltalkHub')
```

Removing Elements

The message remove: anObject removes the first occurrence of an object from the collection. If the collection does not include such an object, it raises an error.

```
ordCol add: 'GitHub'.
ordCol remove: 'GitHub'.
ordCol
>>> an OrderedCollection('Seaside' 'SmalltalkHub' 'GitHub')
```

There is a variant of remove: named remove:ifAbsent: that allows one to specify as second argument a block that is executed in case the element to be removed is not in the collection.

```
result := ordCol remove: 'zork' ifAbsent: [33].
result
>>> 33
```

Conversion

It is possible to get an OrderedCollection from an Array (or any other collection) by sending the message asOrderedCollection:

```
#(1 2 3) asOrderedCollection
>>> an OrderedCollection(1 2 3)
```

```
'hello' asOrderedCollection
>>> an OrderedCollection($h $e $l $l $o)
```

13.8 Interval

The class Interval represents ranges of numbers. For example, the interval of numbers from 1 to 100 is defined as follows:

```
Interval from: 1 to: 100
>>> (1 to: 100)
```

The result of `printString` reveals that the class `Number` provides us with a convenience method called `to:` to generate intervals:

```
(Interval from: 1 to: 100) = (1 to: 100)
>>> true
```

We can use `Interval class>>from:to:by:` or `Number>>to:by:` to specify the step between two numbers as follow:

```
(Interval from: 1 to: 100 by: 0.5) size
>>> 199
```

```
(1 to: 100 by: 0.5) at: 198
>>> 99.5
```

```
(1/2 to: 54/7 by: 1/3) last
>>> (15/2)
```

13.9 Dictionary

Dictionaries are important collections whose elements are accessed using keys. Among the most commonly used messages of dictionary you will find `at: aKey`, `at: aKey put: aValue`, `at: aKey ifAbsent: aBlock`, `keys`, and `values`.

```
| colors |
colors := Dictionary new.
colors at: #yellow put: Color yellow.
colors at: #blue put: Color blue.
colors at: #red put: Color red.
colors at: #yellow
>>> Color yellow
```

```
colors keys
>>> #(#red #blue #yellow)
```

```
colors values
>>> {Color red . Color blue . Color yellow}
```

Dictionaries compare keys by equality. Two keys are considered to be the same if they return true when compared using `=`. A common and difficult to spot bug is to use as key an object whose `=` method has been redefined but not its `hash` method. Both methods are used in the implementation of dictionary and when comparing objects.

In its implementation, a `Dictionary` can be seen as consisting of a set of (key value) associations created using the message `->`. We can create a `Dictionary` from a collection of associations, or we may convert a dictionary to an array of associations.

```
| colors |
colors := Dictionary newFrom: { #blue->Color blue . #red->Color red
    . #yellow->Color yellow }.
colors removeKey: #blue.
colors associations
>>> {#yellow->Color yellow. #red->Color red}
```

13.10 IdentityDictionary

While a dictionary uses the result of the messages = and hash to determine if two keys are the same, the class IdentityDictionary uses the identity (message ==) of the key instead of its values, i.e., it considers two keys to be equal *only* if they are the same object.

Often Symbols are used as keys, in which case it is natural to use an IdentityDictionary, since a Symbol is guaranteed to be globally unique. If, on the other hand, your keys are Strings, it is better to use a plain Dictionary, or you may get into trouble:

```
a := 'foobar'.
b := a copy.
trouble := IdentityDictionary new.
trouble at: a put: 'a'; at: b put: 'b'.
trouble at: a
>>> 'a'
```

```
trouble at: b
>>> 'b'
```

```
trouble at: 'foobar'
>>> 'a'
```

Since a and b are different objects, they are treated as different objects. Interestingly, the literal 'foobar' is allocated just once, so is really the same object as a. You don't want your code to depend on behaviour like this! A plain Dictionary would give the same value for any key equal to 'foobar'.

Use only globally unique objects (like Symbols or SmallIntegers) as keys for an IdentityDictionary, and Strings (or other objects) as keys for a plain Dictionary.

Note that the expression Smalltalk globals returns an instance of SystemDictionary, a subclass of IdentityDictionary, hence all its keys are Symbols (actually, ByteSymbols, which contain only 8-bit characters).

```
Smalltalk globals keys collect: [ :each | each class ] as:Set
>>> a Set(ByteSymbol)
```

Here we are using collect:as: to specify the result collection to be of class Set, that way we collect each kind of class used as a key only once.

13.11 Set

The class `Set` is a collection which behaves as a mathematical set, i.e., as a collection with no duplicate elements and without any order. In a `Set`, elements are added using the message `add:` and they cannot be accessed using the message `at:`. Objects put in a set should implement the methods `hash` and `=`.

```
s := Set new.
s add: 4/2; add: 4; add:2.
s size
>>> 2
```

You can also create sets using `Set class>>newFrom:` or the conversion message `Collection>>asSet`:

```
(Set newFrom: #( 1 2 3 1 4 )) = #(1 2 3 4 3 2 1) asSet
>>> true
```

`asSet` offers us a convenient way to eliminate duplicates from a collection:

```
{ Color black. Color white. (Color red + Color blue + Color green) }
    asSet size
>>> 2
```

Note **red + blue + green = white.**

A `Bag` is much like a `Set` except that it does allow duplicates:

```
{ Color black. Color white. (Color red + Color blue + Color green) }
    asBag size
>>> 3
```

The set operations *union*, *intersection* and *membership test* are implemented by the `Collection` messages `union:`, `intersection:`, and `includes:`. The receiver is first converted to a `Set`, so these operations work for all kinds of collections!

```
(1 to: 6) union: (4 to: 10)
>>> a Set(1 2 3 4 5 6 7 8 9 10)
```

```
'hello' intersection: 'there'
>>> 'eh'
```

```
#Smalltalk includes: $k
>>> true
```

As we explain below, elements of a set are accessed using iterators (see Section 13.14).

13.12 SortedCollection

In contrast to an OrderedCollection, a SortedCollection maintains its elements in sort order. By default, a sorted collection uses the message <= to establish sort order, so it can sort instances of subclasses of the abstract class Magnitude, which defines the protocol of comparable objects (<, =, >, >=, between:and:...). (See Chapter : Basic Classes).

You can create a SortedCollection by creating a new instance and adding elements to it:

```
SortedCollection new add: 5; add: 2; add: 50; add: -10; yourself.
>>> a SortedCollection(-10 2 5 50)
```

More usually, though, one will send the conversion message asSortedCollection to an existing collection:

```
#(5 2 50 -10) asSortedCollection
>>> a SortedCollection(-10 2 5 50)
```

```
'hello' asSortedCollection
>>> a SortedCollection($e $h $l $l $o)
```

How do you get a String back from this result? asString unfortunately returns the printString representation, which is not what we want:

```
'hello' asSortedCollection asString
>>> 'a SortedCollection($e $h $l $l $o)'
```

The correct answer is to either use String class>>newFrom:, String class>>withAll: or Object>>as::

```
'hello' asSortedCollection as: String
>>> 'ehllo'
```

```
String newFrom: 'hello' asSortedCollection
>>> 'ehllo'
```

```
String withAll: 'hello' asSortedCollection
>>> 'ehllo'
```

It is possible to have different kinds of elements in a SortedCollection as long as they are all comparable. For example, we can mix different kinds of numbers such as integers, floats and fractions:

```
{ 5 . 2/ -3 . 5.21 } asSortedCollection
>>> a SortedCollection((-2/3) 5 5.21)
```

Imagine that you want to sort objects that do not define the method <= or that you would like to have a different sorting criterion. You can do this by supplying a two argument block, called a sortblock, to the sorted collection. For example, the class Color is not a Magnitude and it does not implement

the method <=, but we can specify a block stating that the colors should be sorted according to their luminance (a measure of brightness).

```
col := SortedCollection
         sortBlock: [ :c1 :c2 | c1 luminance <= c2 luminance ].
col addAll: { Color red . Color yellow . Color white . Color black }.
col
>>> a SortedCollection(Color black Color red Color yellow Color
    white)
```

13.13 String

In Pharo, a `String` is a collection of `Characters`. It is sequenceable, indexable, mutable and homogeneous, containing only `Character` instances. Like `Arrays`, `Strings` have a dedicated syntax, and are normally created by directly specifying a `String` literal within single quotes, but the usual collection creation methods will work as well.

```
'Hello'
>>> 'Hello'
```

```
String with: $A
>>> 'A'
```

```
String with: $h with: $i with: $!
>>> 'hi!'
```

```
String newFrom: #($h $e $l $l $o)
>>> 'hello'
```

In actual fact, `String` is abstract. When we instantiate a `String` we actually get either an 8-bit `ByteString` or a 32-bit `WideString`. To keep things simple, we usually ignore the difference and just talk about instances of `String`.

While strings are delimited by single quotes, a string can contain a single quote: to define a string with a single quote we should type it twice. Note that the string will contain only one element and not two as shown below:

```
'l''idiot' at: 2
>>> $'
```

```
'l''idiot' at: 3
>>> $i
```

Two or more instances of `String` can be concatenated with a comma.

```
s := 'no', ' ', 'worries'.
s
>>>  'no worries'
```

Since a string is a mutable collection we can also change it using the message `at:put:`.

```
s at: 4 put: $h; at: 5 put: $u.
s
>>> 'no hurries'
```

Note that the comma method is defined by `Collection`, so it will work for any kind of collection!

```
(1 to: 3), '45'
>>> #(1 2 3 $4 $5)
```

We can also modify an existing string using `replaceAll:with:` or `replaceFrom:to:with:` as shown below. Note that the number of characters and the interval should have the same size.

```
s replaceAll: $n with: $N.
s
>>> 'No hurries'
```

```
s replaceFrom: 4 to: 5 with: 'wo'.
s
>>> 'No worries'
```

In contrast to the methods described above, the method `copyReplaceAll:` creates a new string. (Curiously, here the arguments are substrings rather than individual characters, and their sizes do not have to match.)

```
s copyReplaceAll: 'rries' with: 'mbats'
>>> 'No wombats'
```

A quick look at the implementation of these methods reveals that they are defined not only for `Strings`, but for any kind of `SequenceableCollection`, so the following also works:

```
(1 to: 6) copyReplaceAll: (3 to: 5) with: { 'three' . 'etc.' }
>>> #(1 2 'three' 'etc.' 6)
```

String matching

It is possible to ask whether a pattern matches a string by sending the `match:` message. The pattern can use `*` to match an arbitrary series of characters and `#` to match a single character. Note that `match:` is sent to the pattern and not the string to be matched.

```
'Linux *' match: 'Linux mag'
>>> true
```

```
'GNU#Linux #ag' match: 'GNU/Linux tag'
>>> true
```

More advanced pattern matching facilities are also available in the `Regex` package.

Substrings

For substring manipulation we can use messages like `first`, `first:`, `allButFirst:`, `copyFrom:to:` and others, defined in `SequenceableCollection`.

```
'alphabet' at: 6
>>> $b
```

```
'alphabet' first
>>> $a
```

```
'alphabet' first: 5
>>> 'alpha'
```

```
'alphabet' allButFirst: 3
>>> 'habet'
```

```
'alphabet' copyFrom: 5 to: 7
>>> 'abe'
```

```
'alphabet' copyFrom: 3 to: 3
>>> 'p' (not $p)
```

Be aware that result type can be different, depending on the method used. Most of the substring-related methods return `String` instances. But the messages that always return one element of the `String` collection, return a `Character` instance (for example, `'alphabet' at: 6` returns the character $b). For a complete list of substring-related messages, browse the `SequenceableCollection` class (especially the `accessing` protocol).

Some tests on strings

The following examples illustrate the use of `isEmpty`, `includes:` and `anySatisfy:` which are also messages defined not only on Strings but more generally on collections.

```
'Hello' isEmpty
>>> false
```

```
'Hello' includes: $a
>>> false
```

```
'JOE' anySatisfy: [ :c | c isLowercase ]
>>> false
```

```
'Joe' anySatisfy: [ :c | c isLowercase ]
>>> true
```

String templating

There are three messages that are useful to manage string templating: `format:`, `expandMacros` and `expandMacrosWith:`.

```
'{1} is {2}' format: {'Pharo' . 'cool'}
>>> 'Pharo is cool'
```

The messages of the `expandMacros` family offer variable substitution, using `<n>` for carriage return, `<t>` for tabulation, `<1s>`, `<2s>`, `<3s>` for arguments (`<1p>`, `<2p>`, surrounds the string with single quotes), and `<1?value1:value2>` for conditional.

```
'look-<t>-here' expandMacros
>>> 'look-   -here'
```

```
'<1s> is <2s>' expandMacrosWith: 'Pharo' with: 'cool'
>>> 'Pharo is cool'
```

```
'<2s> is <1s>' expandMacrosWith: 'Pharo' with: 'cool'
>>> 'cool is Pharo'
```

```
'<1p> or <1s>' expandMacrosWith: 'Pharo' with: 'cool'
>>> '''Pharo'' or Pharo'
```

```
'<1?Quentin:Thibaut> plays' expandMacrosWith: true
>>> 'Quentin plays'
```

```
'<1?Quentin:Thibaut> plays' expandMacrosWith: false
>>> 'Thibaut plays'
```

Some other utility methods

The class `String` offers numerous other utilities including the messages `asLowercase`, `asUppercase` and `capitalized`.

```
'XYZ' asLowercase
>>> 'xyz'
```

```
'xyz' asUppercase
>>> 'XYZ'
```

```
'hilaire' capitalized
>>> 'Hilaire'
```

```
'Hilaire' uncapitalized
>>> 'hilaire'
```

```
'1.54' asNumber
>>> 1.54
```

```
'this sentence is without a doubt far too long' contractTo: 20
>>> 'this sent...too long'
```

Note that there is generally a difference between asking an object its string representation by sending the message `printString` and converting it to a string by sending the message `asString`. Here is an example of the difference.

```
#ASymbol printString
>>> '#ASymbol'
```

```
#ASymbol asString
>>> 'ASymbol'
```

A symbol is similar to a string but is guaranteed to be globally unique. For this reason symbols are preferred to strings as keys for dictionaries, in particular for instances of `IdentityDictionary`. See also Chapter : Basic Classes for more about `String` and `Symbol`.

13.14 Collection iterators

In Pharo loops and conditionals are simply messages sent to collections or other objects such as integers or blocks (see also Chapter : Understanding message syntax). In addition to low-level messages such as `to:do:` which evaluates a block with an argument ranging from an initial to a final number, the collection hierarchy offers various high-level iterators. Using such iterators will make your code more robust and compact.

Iterating (`do:`)

The method `do:` is the basic collection iterator. It applies its argument (a block taking a single argument) to each element of the receiver. The following example prints all the strings contained in the receiver to the transcript.

```
#('bob' 'joe' 'toto') do: [:each | Transcript show: each; cr].
```

Variants

There are a lot of variants of `do:`, such as `do:without:`, `doWithIndex:` and `reverseDo:`.

For the indexed collections (`Array`, `OrderedCollection`, `SortedCollection`) the message `doWithIndex:` also gives access to the current index. This message is related to `to:do:` which is defined in class `Number`.

```
#('bob' 'joe' 'toto')
    doWithIndex: [ :each :i | (each = 'joe') ifTrue: [ ^ i ] ]
>>> 2
```

For ordered collections, the message `reverseDo:` walks the collection in the reverse order.

The following code shows an interesting message: `do:separatedBy:` which executes the second block only in between two elements.

```
| res |
res := ''.
#('bob' 'joe' 'toto')
   do: [ :e | res := res, e ]
   separatedBy: [ res := res, '.' ].
res
>>> 'bob.joe.toto'
```

Note that this code is not especially efficient since it creates intermediate strings and it would be better to use a write stream to buffer the result (see Chapter : Streams):

```
String streamContents: [ :stream |
   #('bob' 'joe' 'toto') asStringOn: stream delimiter: '.' ]
>>> 'bob.joe.toto'
```

Dictionaries

When the message `do:` is sent to a dictionary, the elements taken into account are the values, not the associations. The proper messages to use are `keysDo:`, `valuesDo:`, and `associationsDo:`, which iterate respectively on keys, values or associations.

```
colors := Dictionary newFrom: { #yellow -> Color yellow. #blue ->
    Color blue. #red -> Color red }.
colors keysDo: [ :key | Transcript show: key; cr ].
colors valuesDo: [ :value | Transcript show: value; cr ].
colors associationsDo: [:value | Transcript show: value; cr].
```

Collecting results (`collect:`)

If you want to apply a function to the elements of a collection and get a new collection with the results, rather than using `do:`, you are probably better off using `collect:`, or one of the other iterator methods. Most of these can be found in the `enumerating` protocol of `Collection` and its subclasses.

Imagine that we want a collection containing the doubles of the elements in another collection. Using the method `do:` we must write the following:

```
| double |
double := OrderedCollection new.
#(1 2 3 4 5 6) do: [ :e | double add: 2 * e ].
double
>>> an OrderedCollection(2 4 6 8 10 12)
```

The message `collect:` executes its argument block for each element and returns a new collection containing the results. Using `collect:` instead, the code is much simpler:

```
#(1 2 3 4 5 6) collect: [ :e | 2 * e ]
>>> #(2 4 6 8 10 12)
```

The advantages of `collect:` over `do:` are even more important in the following example, where we take a collection of integers and generate as a result a collection of absolute values of these integers:

```
aCol :=  #( 2 -3 4 -35 4 -11).
result := aCol species new: aCol size.
1 to: aCol size do: [ :each | result at: each put: (aCol at: each)
    abs ].
result
>>> #(2 3 4 35 4 11)
```

Contrast the above with the much simpler following expression:

```
#( 2 -3 4 -35 4 -11) collect: [ :each | each abs ]
>>> #(2 3 4 35 4 11)
```

A further advantage of the second solution is that it will also work for sets and bags. Generally you should avoid using `do:`, unless you want to send messages to each of the elements of a collection.

Note that sending the message `collect:` returns the same kind of collection as the receiver. For this reason the following code fails. (A `String` cannot hold integer values.)

```
'abc' collect: [ :ea | ea asciiValue ]
>>> "error!"
```

Instead we must first convert the string to an `Array` or an `OrderedCollection`:

```
'abc' asArray collect: [ :ea | ea asciiValue ]
>>> #(97 98 99)
```

Actually `collect:` is not guaranteed to return a collection of exactly the same class as the receiver, but only the same *species*. In the case of an `Interval`, the species is an `Array`!

```
(1 to: 5) collect: [ :ea | ea * 2 ]
>>> #(2 4 6 8 10)
```

Selecting and rejecting elements

The message `select:` returns the elements of the receiver that satisfy a particular condition:

```
(2 to: 20) select: [ :each | each isPrime ]
>>> #(2 3 5 7 11 13 17 19)
```

The message `reject:` does the opposite:

```
(2 to: 20) reject: [ :each | each isPrime ]
>>> #(4 6 8 9 10 12 14 15 16 18 20)
```

Identifying an element with `detect:`

The message `detect:` returns the first element of the receiver that matches block argument.

```
'through' detect: [ :each | each isVowel ]
>>> $o
```

The message `detect:ifNone:` is a variant of the method `detect:`. Its second block is evaluated when there is no element matching the block.

```
Smalltalk globals allClasses
    detect: [:each | '*cobol*' match: each asString]
    ifNone: [ nil ]
>>> nil
```

Accumulating results with `inject:into:`

Functional programming languages often provide a higher-order function called *fold* or *reduce* to accumulate a result by applying some binary operator iteratively over all elements of a collection. In Pharo this is done by `Collection>>inject:into:`.

The first argument is an initial value, and the second argument is a two-argument block which is applied to the result this far, and each element in turn.

A trivial application of `inject:into:` is to produce the sum of a collection of numbers. In Pharo we could write this expression to sum the first 100 integers:

```
(1 to: 100) inject: 0 into: [ :sum :each | sum + each ]
>>> 5050
```

Another example is the following one-argument block which computes factorials:

```
factorial := [ :n | (1 to: n) inject: 1 into: [ :product :each |
    product * each ] ].
factorial value: 10
>>> 3628800
```

Other messages

There are many other iterator messages. You can check the `Collection` class.

count: The message `count:` returns the number of elements satisfying a condition. The condition is represented as a boolean block.

```
Smalltalk globals allClasses
   count: [ :each | 'Collection*' match: each asString ]
>>> 10
```

includes: The message `includes:` checks whether the argument is contained in the collection.

```
| colors |
colors := {Color white . Color yellow .  Color blue . Color orange}.
colors includes: Color blue.
>>> true
```

anySatisfy: The message `anySatisfy:` answers true if at least one element of the collection satisfies the condition represented by the argument.

```
colors anySatisfy: [ :c | c red > 0.5 ]
>>> true
```

13.15 Some hints for using collections

A common mistake with add:

The following error is one of the most frequent Smalltalk mistakes.

```
| collection |
collection := OrderedCollection new add: 1; add: 2.
collection
>>> 2
```

Here the variable `collection` does not hold the newly created collection but rather the last number added. This is because the method `add:` returns the element added and not the receiver.

The following code yields the expected result:

```
| collection |
collection := OrderedCollection new.
collection add: 1; add: 2.
collection
>>> an OrderedCollection(1 2)
```

You can also use the message `yourself` to return the receiver of a cascade of messages:

```
| collection |
collection := OrderedCollection new add: 1; add: 2; yourself
>>> an OrderedCollection(1 2)
```

Listing 13-3 Redefining = and hash.

```
Book >> = aBook
   self class = aBook class ifFalse: [ ^ false ].
   ^ title = aBook title and: [ authors = aBook authors ]

Book >> hash
   ^ title hash bitXor: authors hash
```

Removing an element of the collection you are iterating on

Another mistake you may make is to remove an element from a collection you are currently iterating over.

```
| range |
range := (2 to: 20) asOrderedCollection.
range do: [ :aNumber | aNumber isPrime
                            ifFalse: [ range remove: aNumber ] ].
range
>>> "error!"
```

The solution is to copy the collection before going over it.

```
| range |
range := (2 to: 20) asOrderedCollection.
range copy do: [ :aNumber | aNumber isPrime
                              ifFalse: [ range remove: aNumber ] ].
range
>>> an OrderedCollection(2 3 5 7 11 13 17 19)
```

Redefining = but not hash A difficult error to spot is when you redefine = but not `hash`. The symptoms are that you will lose elements that you put in sets or other strange behaviour. One solution proposed by Kent Beck is to use `bitXor:` to redefine `hash`. Suppose that we want two books to be considered equal if their titles and authors are the same. Then we would redefine not only = but also `hash` as follows:

Another nasty problem arises if you use a mutable object, i.e., an object that can change its hash value over time, as an element of a `Set` or as a key to a `Dictionary`. Don't do this unless you love debugging!

13.16 Chapter summary

The collection hierarchy provides a common vocabulary for uniformly manipulating a variety of different kinds of collections.

- A key distinction is between `SequenceableCollections`, which maintain their elements in a given order, `Dictionary` and its subclasses, which maintain key-to-value associations, and `Sets` and `Bags`, which are unordered.

- You can convert most collections to another kind of collection by sending them the messages `asArray`, `asOrderedCollection`, etc...
- To sort a collection, send it the message `asSortedCollection`.
- `#( ... )` creates arrays containing only literal objects (i.e., objects created without sending messages). `{ ... }` creates dynamic arrays using a compact form.
- A `Dictionary` compares keys by equality. It is most useful when keys are instances of `String`. An `IdentityDictionary` instead uses object identity to compare keys. It is more suitable when `Symbols` are used as keys, or when mapping object references to values.
- `Strings` also understand the usual collection messages. In addition, a `String` supports a simple form of pattern-matching. For more advanced application, look instead at the RegEx package.
- The basic iteration message is `do:`. It is useful for imperative code, such as modifying each element of a collection, or sending each element a message.
- Instead of using `do:`, it is more common to use `collect:`, `select:`, `reject:`, `includes:`, `inject:into:` and other higher-level messages to process collections in a uniform way.
- Never remove an element from a collection you are iterating over. If you must modify it, iterate over a copy instead.
- If you override `=`, remember to override `hash` as well!

CHAPTER 14

Streams

Streams are used to iterate over sequences of elements such as sequenced collections, files, and network streams. Streams may be either readable, or writeable, or both. Reading or writing is always relative to the current position in the stream. Streams can easily be converted to collections, and vice versa.

14.1 Two sequences of elements

A good metaphor to understand a stream is the following. A stream can be represented as two sequences of elements: a past element sequence and a future element sequence. The stream is positioned between the two sequences. Understanding this model is important, since all stream operations in Pharo rely on it. For this reason, most of the `Stream` classes are subclasses of `PositionableStream`. Figure 14-1 presents a stream which contains five characters. This stream is in its original position, i.e., there is no element in the past. You can go back to this position using the message `reset` defined in `PositionableStream`.

Reading an element conceptually means removing the first element of the future element sequence and putting it after the last element in the past ele-

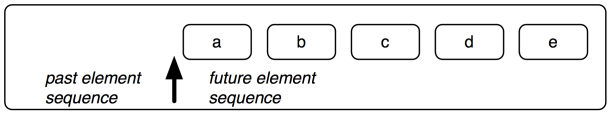

Figure 14-1 A stream positioned at its beginning.

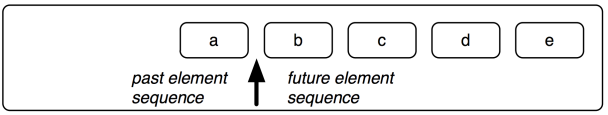

Figure 14-2 The same stream after the execution of the method `next`: the character a is *in the past* whereas b, c, d and e are *in the future*.

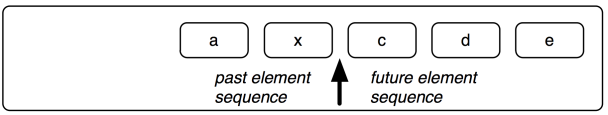

Figure 14-3 The same stream after having written an x.

ment sequence. After having read one element using the message `next`, the state of your stream is that shown in Figure 14-2.

Writing an element means replacing the first element of the future sequence by the new one and moving it to the past. Figure 14-3 shows the state of the same stream after having written an x using the message `nextPut: anElement` defined in `Stream`.

14.2 Streams vs. collections

The collection protocol supports the storage, removal and enumeration of the elements of a collection, but does not allow these operations to be intermingled. For example, if the elements of an `OrderedCollection` are processed by a `do:` method, it is not possible to add or remove elements from inside the `do:` block. Nor does the collection protocol offer ways to iterate over two collections at the same time, choosing which collection goes forward and which does not. Procedures like these require that a traversal index or position reference is maintained outside of the collection itself: this is exactly the role of `ReadStream`, `WriteStream` and `ReadWriteStream`.

These three classes are defined to *stream over* some collection. For example, the following snippet creates a stream on an interval, then it reads two elements.

```
| r |
r := ReadStream on: (1 to: 1000).
r next.
>>> 1
```

```
r next.
>>> 2
```

```
r atEnd.
>>> false
```

`WriteStreams` can write data to the collection:

```
| w |
w := WriteStream on: (String new: 5).
w nextPut: $a.
w nextPut: $b.
w contents.
>>>  'ab'
```

It is also possible to create `ReadWriteStreams` that support both the reading and writing protocols.

The following sections present the protocols in more depth.

14.3 Streaming over collections

Streams are really useful when dealing with collections of elements, and can be used for reading and writing those elements. We will now explore the stream features for collections.

Reading collections

Using a stream to read a collection essentially provides you a pointer into the collection. That pointer will move forward on reading, and you can place it wherever you want. The class `ReadStream` should be used to read elements from collections.

Messages `next` and `next:` defined in `ReadStream` are used to retrieve one or more elements from the collection.

```
| stream |
stream := ReadStream on: #(1 (a b c) false).
stream next.
>>>   1
```

```
stream next.
>>>   #(#a #b #c)
```

```
stream next.
>>>   false
```

```
| stream |
stream := ReadStream on: 'abcdef'.
stream next: 0.
>>>   ''
```

```
stream next: 1.
>>>   'a'
```

```
stream next: 3.
>>>   'bcd'
```

```
stream next: 2.
>>>   'ef'
```

The message `peek` defined in `PositionableStream` is used when you want to know what is the next element in the stream without going forward.

```
| stream negative number |
stream := ReadStream on: '-143'.
"look at the first element without consuming it."
negative := (stream peek = $-).
negative.
>>> true
```

```
"ignores the minus character"
negative ifTrue: [ stream next ].
number := stream upToEnd.
number.
>>> '143'
```

This code sets the boolean variable `negative` according to the sign of the number in the stream, and `number` to its absolute value. The message `upToEnd` defined in `ReadStream` returns everything from the current position to the end of the stream and sets the stream to its end. This code can be simplified using the message `peekFor:` defined in `PositionableStream`, which moves forward if the following element equals the parameter and doesn't move otherwise.

```
| stream |
stream := '-143' readStream.
(stream peekFor: $-).
>>> true
```

```
stream upToEnd
>>> '143'
```

`peekFor:` also returns a boolean indicating if the parameter equals the element.

You might have noticed a new way of constructing a stream in the above example: one can simply send the message `readStream` to a sequenceable collection (such as a `String`) to get a reading stream on that particular collection.

14.4 Positioning

There are messages to position the stream pointer. If you have the index, you can go directly to it using `position:` defined in `PositionableStream`. You can request the current position using `position`. Please remember that

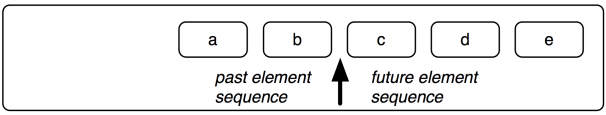

Figure 14-4 A stream at position 2.

a stream is not positioned on an element, but between two elements. The index corresponding to the beginning of the stream is 0.

You can obtain the state of the stream depicted in 14-4 with the following code:

```
| stream |
stream := 'abcde' readStream.
stream position: 2.
stream peek
>>> $c
```

To position the stream at the beginning or the end, you can use the message `reset` or `setToEnd`. The messages `skip:` and `skipTo:` are used to go forward to a location relative to the current position: `skip:` accepts a number as argument and skips that number of elements whereas `skipTo:` skips all elements in the stream until it finds an element equal to its parameter. Note that it positions the stream after the matched element.

```
| stream |
stream := 'abcdef' readStream.
stream next.
>>> $a    "stream is now positioned just after the a"
```

```
stream skip: 3.                          "stream is now after the d"
stream position.
>>>   4
```

```
stream skip: -2.                         "stream is after the b"
stream position.
>>> 2
```

```
stream reset.
stream position.
>>> 0
```

```
stream skipTo: $e.                      "stream is just after the e
    now"
stream next.
>>> $f
```

```
stream contents.
>>> 'abcdef'
```

As you can see, the letter e has been skipped.

```
| stream1 stream2 result |
stream1 := #(1 4 9 11 12 13) readStream.
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.

"The variable result will contain the sorted collection."
result := OrderedCollection new.
[stream1 atEnd not & stream2 atEnd not ]
  whileTrue: [
    stream1 peek < stream2 peek
        "Remove the smallest element from either stream and add it
    to the result."
        ifTrue: [result add: stream1 next ]
        ifFalse: [result add: stream2 next ] ].

"One of the two streams might not be at its end. Copy whatever
    remains."
result
  addAll: stream1 upToEnd;
  addAll: stream2 upToEnd.

result.
>>>   an OrderedCollection(1 1 2 3 4 4 5 9 10 11 12 13 13 14 15)
```

The message `contents` always returns a copy of the entire stream.

14.5 Testing

Some messages allow you to test the state of the current stream: `atEnd` returns `true` if and only if no more elements can be read, whereas `isEmpty` returns `true` if and only if there are no elements at all in the collection.

Here is a possible implementation of an algorithm using `atEnd` that takes two sorted collections as parameters and merges those collections into another sorted collection:

14.6 Writing to collections

We have already seen how to read a collection by iterating over its elements using a `ReadStream`. We'll now learn how to create collections using `WriteStreams`.

`WriteStreams` are useful for appending a lot of data to a collection at various locations. They are often used to construct strings that are based on static and dynamic parts, as in this example:

This technique is used in the different implementations of the method `printOn:`, for example. There is a simpler and more efficient way of creating strings if you are only interested in the content of the stream:

```
| stream |
stream := String new writeStream.
stream
  nextPutAll: 'This Smalltalk image contains: ';
  print: Smalltalk allClasses size;
  nextPutAll: ' classes.';
  cr;
  nextPutAll: 'This is really a lot.'.

stream contents.
>>> 'This Smalltalk image contains: 9003 classes.
This is really a lot.'
```

```
| string |
string := String streamContents:
    [ :stream |
      stream
                print: #(1 2 3);
                space;
                nextPutAll: 'size';
                space;
                nextPut: $=;
                space;
                print: 3.   ].
string.
>>>   '#(1 2 3) size = 3'
```

The message `streamContents:` defined `SequenceableCollection` creates a collection and a stream on that collection for you. It then executes the block you gave passing the stream as a parameter. When the block ends, `streamContents:` returns the contents of the collection.

The following `WriteStream` methods are especially useful in this context:

`nextPut:` adds the parameter to the stream;

`nextPutAll:` adds each element of the collection, passed as a parameter, to the stream;

`print:` adds the textual representation of the parameter to the stream.

There are also convenient messages for printing useful characters to a stream, such as `space`, `tab` and `cr` (carriage return). Another useful method is `ensureASpace` which ensures that the last character in the stream is a space; if the last character isn't a space it adds one.

```
[| temp |
  temp := String new.
  (1 to: 100000)
    do: [:i | temp := temp, i asString, ' ' ] ] timeToRun
>>> 0:00:01:54.758
```

```
String streamContents: [ :tempStream |
  (1 to: 100000)
       do: [:i | tempStream nextPutAll: i asString; space ] ]
```

14.7 About String Concatenation

Using `nextPut:` and `nextPutAll:` on a `WriteStream` is often the best way to concatenate characters. Using the comma concatenation operator (`,`) is far less efficient:

```
[| temp |
  temp := WriteStream on: String new.
  (1 to: 100000)
    do: [:i | temp nextPutAll: i asString; space ].
  temp contents ] timeToRun
>>> 0:00:00:00.024
```

The reason that using a stream can be much more efficient is that using a comma creates a new string containing the concatenation of the receiver and the argument, so it must copy both of them. When you repeatedly concatenate onto the same receiver, it gets longer and longer each time, so that the number of characters that must be copied goes up exponentially. This also creates a lot of garbage, which must be collected. Using a stream instead of string concatenation is a well-known optimization.

In fact, you can use the message `streamContents:` defined in `SequenceableCollection class` (mentioned earlier) to help you do this:

14.8 Reading and writing at the same time

It's possible to use a stream to access a collection for reading and writing at the same time. Imagine you want to create a `History` class which will manage backward and forward buttons in a web browser. A history would react as in figures 14-10 to 14-16.

This behaviour can be implemented using a `ReadWriteStream`.

Nothing really difficult here, we define a new class which contains a stream. The stream is created during the `initialize` method.

We need methods to go backward and forward:

Figure 14-10 A new history is empty. Nothing is displayed in the web browser.

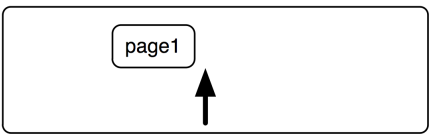

Figure 14-11 The user opens to page 1.

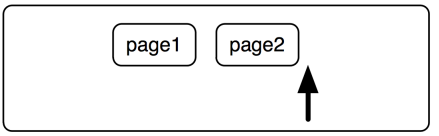

Figure 14-12 The user clicks on a link to page 2.

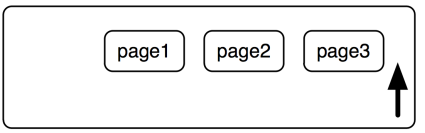

Figure 14-13 The user clicks on a link to page 3.

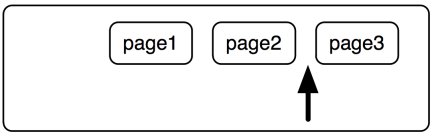

Figure 14-14 The user clicks on the Back button. They are now viewing page 2 again.

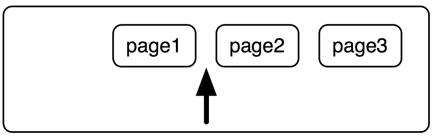

Figure 14-15 The user clicks again the back button. Page 1 is now displayed.

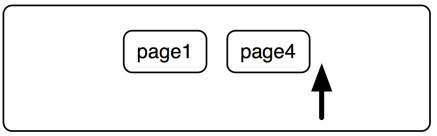

Figure 14-16 From page 1, the user clicks on a link to page 4. The history forgets pages 2 and 3.

```
Object subclass: #History
  instanceVariableNames: 'stream'
  classVariableNames: ''
  package: 'PBE-Streams'

History >> initialize
    super initialize.
    stream := ReadWriteStream on: Array new.
```

```
History >> goBackward
  self canGoBackward
    ifFalse: [ self error: 'Already on the first element' ].
  stream skip: -2.
  ^ stream next.

History >> goForward
  self canGoForward
    ifFalse: [ self error: 'Already on the last element' ].
  ^ stream next
```

Up to this point, the code is pretty straightforward. Next, we have to deal with the `goTo:` method which should be activated when the user clicks on a link. A possible implementation is:

This version is incomplete however. This is because when the user clicks on the link, there should be no more future pages to go to, *i.e.*, the forward button must be deactivated. To do this, the simplest solution is to write `nil` just after, to indicate that history is at the end:

Now, only methods `canGoBackward` and `canGoForward` remain to be implemented.

A stream is always positioned between two elements. To go backward, there must be two pages before the current position: one page is the current page, and the other one is the page we want to go to.

Let us add a method to peek at the contents of the stream:

```
History >> goTo: aPage
    stream nextPut: aPage.
```

```
History >> goTo: anObject
  stream nextPut: anObject.
  stream nextPut: nil.
  stream back.
```

```
History >> canGoBackward
  ^ stream position > 1

History >> canGoForward
  ^ stream atEnd not and: [stream peek notNil ]
```

```
History >> contents
  ^ stream contents
```

And the history works as advertised:

14.9 Chapter summary

Streams offer a better way (compared to collections) to incrementally read and write a sequence of elements. There are easy ways to convert back and forth between streams and collections.

- Streams may be either readable, writeable or both readable and writeable.
- To convert a collection to a stream, define a stream *on* a collection, *e.g.*, `ReadStream on: (1 to: 1000)`, or send the messages `readStream`, etc. to the collection.
- To convert a stream to a collection, send the message `contents`.
- To concatenate large collections, instead of using the comma operator, it is more efficient to create a stream, append the collections to the stream with `nextPutAll:`, and extract the result by sending `contents`.

```
History new
  goTo: #page1;
  goTo: #page2;
  goTo: #page3;
  goBackward;
  goBackward;
  goTo: #page4;
  contents
>>> #(#page1 #page4 nil nil)
```

CHAPTER 15

Morphic

Morphic is the name given to Pharo's graphical interface. Morphic supports two main aspects: on one hand Morphic defines all the low-level graphical entities and related infrastructure (events, drawing,...) and on the other hand Morphic defines all the widgets available in Pharo. Morphic is written in Pharo, so it is fully portable between operating systems. As a consequence, Pharo looks exactly the same on Unix, MacOS and Windows. What distinguishes Morphic from most other user interface toolkits is that it does not have separate modes for *composing* and *running* the interface: all the graphical elements can be assembled and disassembled by the user, at any time. We thank Hilaire Fernandes for permission to base this chapter on his original article in French.

15.1 The history of Morphic

Morphic was developed by John Maloney and Randy Smith for the Self programming language, starting around 1993. Maloney later wrote a new version of Morphic for Squeak, but the basic ideas behind the Self version are still alive and well in Pharo Morphic: *directness* and *liveness*. Directness means that the shapes on the screen are objects that can be examined or changed directly, that is, by clicking on them using a mouse. Liveness means that the user interface is always able to respond to user actions: information on the screen is continuously updated as the world that it describes changes. A simple example of this is that you can detach a menu item and keep it as a button.

Bring up the World Menu and Option-Command-Shift click once on it to bring up its morphic halo, then repeat the operation again on a menu item you want to detach, to bring up that item's halo (see Figure 15-1).

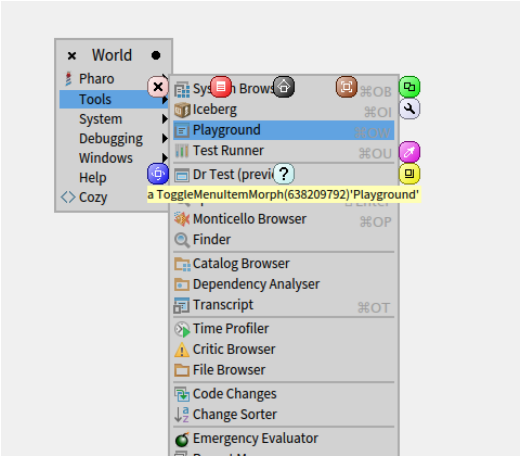

Figure 15-1 Detaching a morph, here the `Playground` menu item, to make it an independent button.

Figure 15-2 Dropping the menu item on the desktop, here the `Playground` menu item is now an independent button.

Now drag that item elsewhere on the screen by grabbing the black handle (which looks like a pliers), as shown in Figure 15-1.

Once dropped the menu item stays detached from the menu and you can interact with it as if it would be in the menu (see Figurefig:detachingMenu2).

This example illustrates what we mean by *directness* and *liveness*. This gives a lot of power when developing alternate user interface and prototyping alternate interactions. Some people

Listing 15-3 Creation of a String Morph

```
'Morph' asMorph openInWorld
```

Listing 15-4 Getting a morph for an instance of `Color`

```
Color >> asMorph
  ^ Morph new color: self
```

Morphic is a bit showing its age, and the Pharo community is working since several years on a possible replacement. Replacing Morphic means to have both a new low-level infrastructure and a new widget sets. The project is called Bloc and got several iterations. Bloc is about the infrastructure and Brick is a set of widgets built on top. But let us have fun with Morphic.

15.2 Morphs

All of the objects that you see on the screen when you run Pharo are *Morphs*, that is, they are instances of subclasses of class `Morph`. The class `Morph` itself is a large class with many methods; this makes it possible for subclasses to implement interesting behaviour with little code. You can create a morph to represent any object, although how good a representation you get depends on the object!

To create a morph to represent a string object, execute the following code in a Playground.

This creates a Morph to represent the string `'Morph'`, and then opens it (that is, displays it) in the *world*, which is the name that Pharo gives to the screen. You should obtain a graphical element (a `Morph`), which you can manipulate by meta-clicking.

Of course, it is possible to define morphs that are more interesting graphical representations than the one that you have just seen. The method `asMorph` has a default implementation in class `Object` class that just creates a `StringMorph`. So, for example, `Color tan asMorph` returns a StringMorph labeled with the result of `Color tan printString`. Let's change this so that we get a coloured rectangle instead.

Open a browser on the `Color` class and add the following method to it:

Now execute `Color orange asMorph openInWorld` in a Playground. Instead of the string-like morph, you get an orange rectangle (see Figure 15-5)! You get the same executing `(Morph new color: Color orange) openInWorld`

Figure 15-5 `(Morph new color: Color orange) openInWorld` or `Color orange asMorph openInWorld` with our new method.

15.3 Manipulating morphs

Morphs are objects, so we can manipulate them like any other object in Pharo: by sending messages, we can change their properties, create new subclasses of Morph, and so on.

Every morph, even if it is not currently open on the screen, has a position and a size. For convenience, all morphs are considered to occupy a rectangular region of the screen; if they are irregularly shaped, their position and size are those of the smallest rectangular *box* that surrounds them, which is known as the morph's bounding box, or just its *bounds*. The `position` method returns a `Point` that describes the location of the morph's upper left corner (or the upper left corner of its bounding box). The origin of the coordinate system is the screen's upper left corner, with *y* coordinates increasing *down* the screen and *x* coordinates increasing to the right. The `extent` method also returns a point, but this point specifies the width and height of the morph rather than a location.

Type the following code into a playground and `Do it`:

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red.
bill openInWorld.
```

Then type `joe position` and then `Print it`. To move joe, execute `joe position: (joe position + (10@3))` repeatedly (see Figure 15-6).

It is possible to do a similar thing with size. `joe extent` answers joe's size; to have joe grow, execute `joe extent: (joe extent * 1.1)`. To change the color of a morph, send it the `color:` message with the desired `Color`

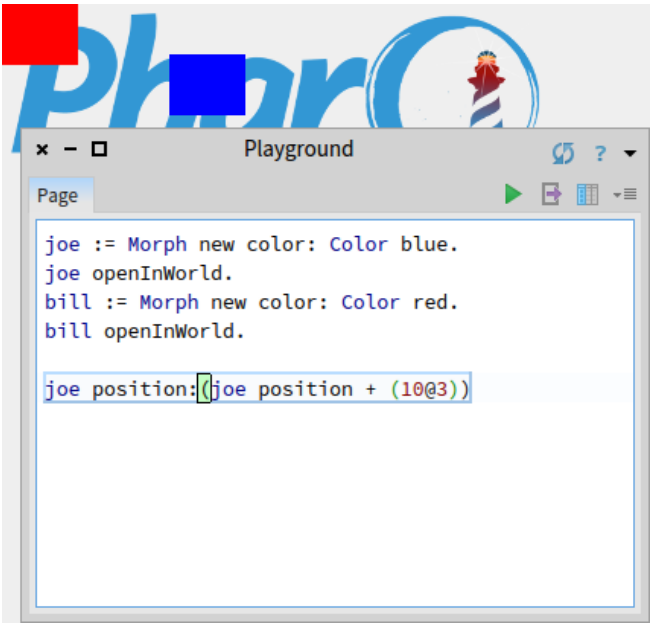

Figure 15-6 Bill and Joe after 10 moves.

object as argument, for instance, `joe color: Color orange`. To add transparency, try `joe color: (Color orange alpha: 0.5)`.

To make bill follow joe, you can repeatedly execute this code:

```
bill position: (joe position + (100@0))
```

If you move `joe` using the mouse and then execute this code, bill will move so that it is 100 pixels to the right of `joe`. You can see the result on Figure 15-7. Nothing suprising.

15.4 Composing morphs

One way of creating new graphical representations is by placing one morph inside another. This is called *composition*; morphs can be composed to any depth. You can place a morph inside another by sending the message `addMorph:` to the container morph.

Try adding a morph to another one as follows:

```
balloon := BalloonMorph new color: Color yellow.
joe addMorph: balloon.
balloon position: joe position.
```

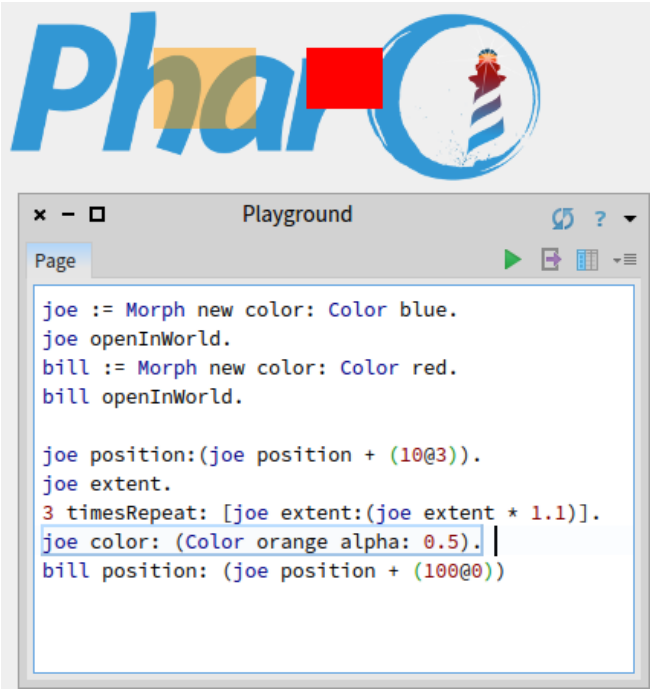

Figure 15-7 Bill follows Joe.

Figure 15-8 The balloon is contained inside joe, the translucent orange morph.

The last line positions the balloon at the same coordinates as joe. Notice that the coordinates of the contained morph are still relative to the screen, not to the containing morph. This absolute way of positioning morph is not really good and it makes programming morphs feels a bit odd. But there are many methods available to position a morph; browse the `geometry` protocol of class `Morph` to see for yourself. For example, to center the balloon inside joe, execute `balloon center: joe center`.

If you now try to grab the balloon with the mouse, you will find that you actually grab joe, and the two morphs move together: the balloon is *embedded* inside joe. It is possible to embed more morphs inside joe. In addition to doing this programmatically, you can also embed morphs by direct manipulation.

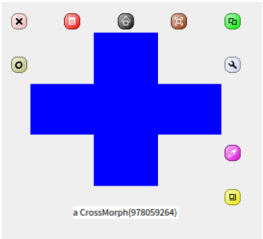

Figure 15-9 A `CrossMorph` with its halo; you can resize it as you wish.

15.5 Creating and drawing your own morphs

While it is possible to make many interesting and useful graphical representations by composing morphs, sometimes you will need to create something completely different.

To do this you define a subclass of `Morph` and override the `drawOn:` method to change its appearance.

The morphic framework sends the message `drawOn:` to a morph when it needs to redisplay the morph on the screen. The parameter to `drawOn:` is a kind of `Canvas`; the expected behaviour is that the morph will draw itself on that canvas, inside its bounds. Let's use this knowledge to create a cross-shaped morph.

Using the browser, define a new class `CrossMorph` inheriting from `Morph`:

```
Morph subclass: #CrossMorph
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-Morphic'
```

We can define the `drawOn:` method like this:

```
CrossMorph >> drawOn: aCanvas
  | crossHeight crossWidth horizontalBar verticalBar |
  crossHeight := self height / 3.
  crossWidth := self width / 3.
  horizontalBar := self bounds insetBy: 0 @ crossHeight.
  verticalBar := self bounds insetBy: crossWidth @ 0.
  aCanvas fillRectangle: horizontalBar color: self color.
  aCanvas fillRectangle: verticalBar color: self color
```

Sending the `bounds` message to a morph answers its bounding box, which is an instance of `Rectangle`. Rectangles understand many messages that create other rectangles of related geometry. Here, we use the `insetBy:` message with a point as its argument to create first a rectangle with reduced height, and then another rectangle with reduced width.

To test your new morph, execute `CrossMorph new openInWorld`.

The result should look something like Figure 15-9. However, you will notice that the sensitive zone — where you can click to grab the morph — is still the whole bounding box. Let's fix this.

When the Morphic framework needs to find out which Morphs lie under the cursor, it sends the message `containsPoint:` to all the morphs whose bounding boxes lie under the mouse pointer. So, to limit the sensitive zone of the morph to the cross shape, we need to override the `containsPoint:` method.

Define the following method in class `CrossMorph`:

```
CrossMorph >> containsPoint: aPoint
  | crossHeight crossWidth horizontalBar verticalBar |
  crossHeight := self height / 3.
  crossWidth := self width / 3.
  horizontalBar := self bounds insetBy: 0 @ crossHeight.
  verticalBar := self bounds insetBy: crossWidth @ 0.
  ^ (horizontalBar containsPoint: aPoint) or: [ verticalBar
    containsPoint: aPoint ]
```

This method uses the same logic as `drawOn:`, so we can be confident that the points for which `containsPoint:` answers `true` are the same ones that will be colored in by `drawOn`. Notice how we leverage the `containsPoint:` method in class `Rectangle` to do the hard work.

There are two problems with the code in the two methods above.

The most obvious is that we have duplicated code. This is a cardinal error: if we find that we need to change the way that `horizontalBar` or `verticalBar` are calculated, we are quite likely to forget to change one of the two occurrences. The solution is to factor out these calculations into two new methods, which we put in the `private` protocol:

```
CrossMorph >> horizontalBar
  | crossHeight |
  crossHeight := self height / 3.
  ^ self bounds insetBy: 0 @ crossHeight
```

```
CrossMorph >> verticalBar
  | crossWidth |
  crossWidth := self width / 3.
  ^ self bounds insetBy: crossWidth @ 0
```

We can then define both `drawOn:` and `containsPoint:` using these methods:

```
CrossMorph >> drawOn: aCanvas
  aCanvas fillRectangle: self horizontalBar color: self color.
  aCanvas fillRectangle: self verticalBar color: self color
```

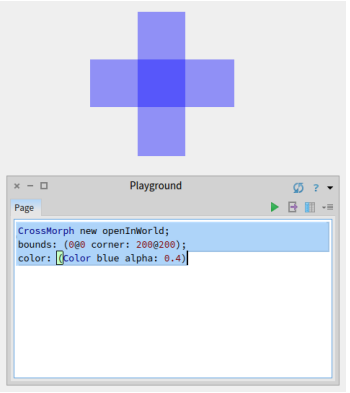

Figure 15-10 The center of the cross is filled twice with the color.

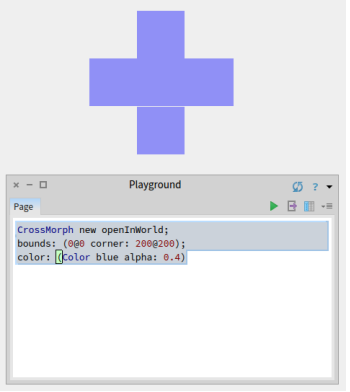

Figure 15-11 The cross-shaped morph, showing a row of unfilled pixels.

```
CrossMorph >> containsPoint: aPoint
  ^ (self horizontalBar containsPoint: aPoint) or: [ self
    verticalBar containsPoint: aPoint ]
```

This code is much simpler to understand, largely because we have given meaningful names to the private methods. In fact, it is so simple that you may have noticed the second problem: the area in the center of the cross, which is under both the horizontal and the vertical bars, is drawn twice. This doesn't matter when we fill the cross with an opaque colour, but the bug becomes apparent immediately if we draw a semi-transparent cross, as shown in Figure 15-10.

Execute the following code in a playground:

```
CrossMorph new openInWorld;
   bounds: (0@0 corner: 200@200);
   color: (Color blue alpha: 0.4)
```

The fix is to divide the vertical bar into three pieces, and to fill only the top and bottom. Once again we find a method in class `Rectangle` that does the hard work for us: `r1 areasOutside: r2` answers an array of rectangles comprising the parts of `r1` outside `r2`. Here is the revised code:

```
CrossMorph >> drawOn: aCanvas
  | topAndBottom |
  aCanvas fillRectangle: self horizontalBar color: self color.
  topAndBottom := self verticalBar areasOutside: self horizontalBar.
  topAndBottom do: [ :each | aCanvas fillRectangle: each color: self
    color ]
```

This code seems to work, but if you try it on some crosses and resize them, you may notice that at some sizes, a one-pixel wide line separates the bottom of the cross from the remainder, as shown in Figure 15-11. This is due to rounding: when the size of the rectangle to be filled is not an integer, `fillRectangle: color:` seems to round inconsistently, leaving one row of pixels unfilled.

We can work around this by rounding explicitly when we calculate the sizes of the bars as shown hereafter:

```
CrossMorph >> horizontalBar
  | crossHeight |
  crossHeight := (self height / 3) rounded.
  ^ self bounds insetBy: 0 @ crossHeight
```

```
CrossMorph >> verticalBar
  | crossWidth |
  crossWidth := (self width / 3) rounded.
  ^ self bounds insetBy: crossWidth @ 0
```

15.6 Mouse events for interaction

To build live user interfaces using morphs, we need to be able to interact with them using the mouse and keyboard. Moreover, the morphs need to be able respond to user input by changing their appearance and position — that is, by animating themselves.

When a mouse button is pressed, Morphic sends each morph under the mouse pointer the message `handlesMouseDown:`. If a morph answers `true`, then Morphic immediately sends it the `mouseDown:` message; it also sends the `mouseUp:` message when the user releases the mouse button. If all morphs answer `false`, then Morphic initiates a drag-and-drop operation. As we will discuss below, the `mouseDown:` and `mouseUp:` messages are sent with an ar-

gument — a `MouseEvent` object — that encodes the details of the mouse action.

Let's extend `CrossMorph` to handle mouse events. We start by ensuring that all crossMorphs answer `true` to the `handlesMouseDown:` message. Add the method to `CrossMorph` defined as follows:

```
CrossMorph >> handlesMouseDown: anEvent
  ^ true
```

Suppose that when we click on the cross, we want to change the color of the cross to red, and when we action-click on it, we want to change the color to yellow. We define the `mouseDown:` method as follows:

```
CrossMorph >> mouseDown: anEvent
  anEvent redButtonPressed
    ifTrue: [ self color: Color red ].  "click"
  anEvent yellowButtonPressed
    ifTrue: [ self color: Color yellow ]. "action-click"
  self changed
```

Notice that in addition to changing the color of the morph, this method also sends `self changed`. This makes sure that morphic sends `drawOn:` in a timely fashion.

Note also that once the morph handles mouse events, you can no longer grab it with the mouse and move it. Instead you have to use the halo: Option-Command-Shift click on the morph to make the halo appear and grab either the brown move handle or the black pickup handle at the top of the morph.

The `anEvent` argument of `mouseDown:` is an instance of `MouseEvent`, which is a subclass of `MorphicEvent`. `MouseEvent` defines the `redButtonPressed` and `yellowButtonPressed` methods. Browse this class to see what other methods it provides to interrogate the mouse event.

15.7 Keyboard events

To catch keyboard events, we need to take three steps.

1. Give the *keyboard focus* to a specific morph. For instance, we can give focus to our morph when the mouse is over it.
2. Handle the keyboard event itself with the `handleKeystroke:` method. This message is sent to the morph that has keyboard focus when the user presses a key.
3. Release the keyboard focus when the mouse is no longer over our morph.

Let's extend `CrossMorph` so that it reacts to keystrokes. First, we need to arrange to be notified when the mouse is over the morph. This will happen if our morph answers `true` to the `handlesMouseOver:` message.

```
CrossMorph >> handlesMouseOver: anEvent
  ^ true
```

This message is the equivalent of `handlesMouseDown:` for the mouse position. When the mouse pointer enters or leaves the morph, the `mouseEnter:` and `mouseLeave:` messages are sent to it.

Define two methods so that `CrossMorph` catches and releases the keyboard focus, and a third method to actually handle the keystrokes.

```
CrossMorph >> mouseEnter: anEvent
  anEvent hand newKeyboardFocus: self
```

```
CrossMorph >> mouseLeave: anEvent
  anEvent hand newKeyboardFocus: nil
```

```
CrossMorph >> handleKeystroke: anEvent
  | keyValue |
  keyValue := anEvent keyValue.
  keyValue = 30  "up arrow"
    ifTrue: [self position: self position - (0 @ 1)].
  keyValue = 31  "down arrow"
    ifTrue: [self position: self position + (0 @ 1)].
  keyValue = 29  "right arrow"
    ifTrue: [self position: self position + (1 @ 0)].
  keyValue = 28  "left arrow"
    ifTrue: [self position: self position - (1 @ 0)]
```

We have written this method so that you can move the morph using the arrow keys. Note that when the mouse is no longer over the morph, the `handleKeystroke:` message is not sent, so the morph stops responding to keyboard commands. To discover the key values, you can open a Transcript window and add `Transcript show: anEvent keyValue` to the `handleKeystroke:` method.

The `anEvent` argument of `handleKeystroke:` is an instance of `KeyboardEvent`, another subclass of `MorphicEvent`. Browse this class to learn more about keyboard events.

15.8 Morphic animations

Morphic provides a simple animation system with two main methods: `step` is sent to a morph at regular intervals of time, while `stepTime` specifies the time in milliseconds between `steps`. `stepTime` is actually the *minimum* time between `steps`. If you ask for a `stepTime` of 1 ms, don't be surprised if Pharo is too busy to step your morph that often. In addition, `startStepping` turns on the stepping mechanism, while `stopStepping` turns it off again. `isStepping` can be used to find out whether a morph is currently being stepped.

Make `CrossMorph` blink by defining these methods as follows:

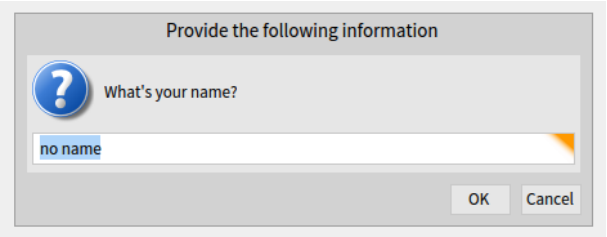

Figure 15-12 An input dialog.

```
CrossMorph >> stepTime
  ^ 100
```

```
CrossMorph >> step
  (self color diff: Color black) < 0.1
    ifTrue: [ self color: Color red ]
    ifFalse: [ self color: self color darker ]
```

To start things off, you can open an inspector on a `CrossMorph` using the debug handle which look like a wrench in the morphic halo, type `self startStepping` in the small pane at the bottom, and `Do it`.

You can also redefine the `initialize` method as follows:

```
CrossMorph >> initialize
  super initialize.
  self startStepping
```

Alternatively, you can modify the `handleKeystroke:` method so that you can use the `+` and `-` keys to start and stop stepping. Add the following code to the `handleKeystroke:` method:

```
  keyValue = $+ asciiValue
    ifTrue: [ self startStepping ].
  keyValue = $- asciiValue
    ifTrue: [ self stopStepping ]
```

15.9 Interactors

To prompt the user for input, the `UIManager` class provides a large number of ready to use dialog boxes. For instance, the `request:initialAnswer:` method returns the string entered by the user (Figure 15-12).

```
UIManager default request: 'What''s your name?' initialAnswer: 'no
    name'
```

To display a popup menu, use one of the various `chooseFrom:` methods (Figure 15-13):

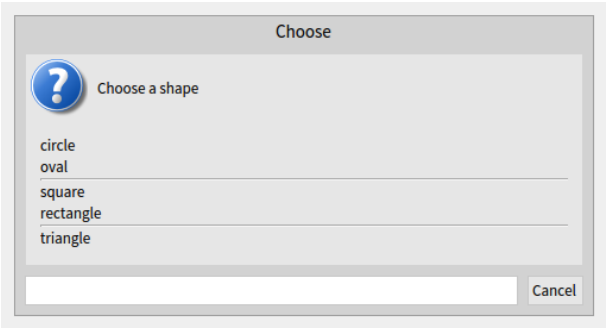

Figure 15-13 Pop-up menu.

```
UIManager default
  chooseFrom: #('circle' 'oval' 'square' 'rectangle' 'triangle')
  lines: #(2 4) message: 'Choose a shape'
```

Browse the `UIManager` class and try out some of the interaction methods offered.

15.10 Drag-and-drop

Morphic also supports drag-and-drop. Let's examine a simple example with two morphs, a receiver morph and a dropped morph. The receiver will accept a morph only if the dropped morph matches a given condition: in our example, the morph should be blue. If it is rejected, the dropped morph decides what to do.

Let's first define the receiver morph:

```
Morph subclass: #ReceiverMorph
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-Morphic'
```

Now define the initialization method in the usual way:

```
ReceiverMorph >> initialize
  super initialize.
  color := Color red.
  bounds := 0 @ 0 extent: 200 @ 200
```

How do we decide if the receiver morph will accept or reject the dropped morph? In general, both of the morphs will have to agree to the interaction. The receiver does this by responding to `wantsDroppedMorph:event:`. Its first argument is the dropped morph, and the second the mouse event, so that the receiver can, for example, see if any modifier keys were held down

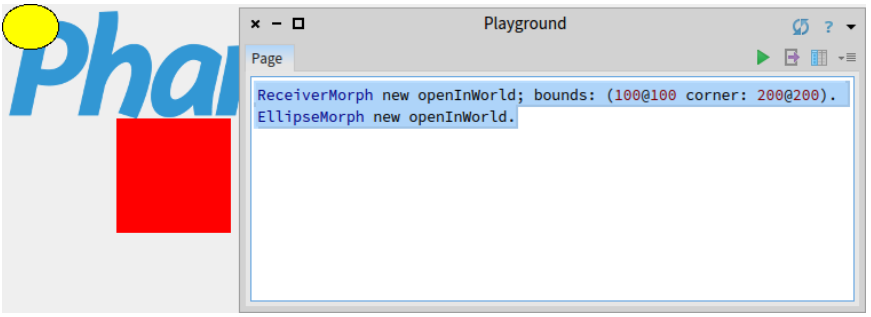

Figure 15-14 A `ReceiverMorph` and an `EllipseMorph`.

at the time of the drop. The dropped morph is also given the opportunity to check and see if it likes the morph onto which it is being dropped, by responding to the message `wantsToBeDroppedInto:`. The default implementation of this method (in class `Morph`) answers `true`.

```
ReceiverMorph >> wantsDroppedMorph: aMorph event: anEvent
  ^ aMorph color = Color blue
```

What happens to the dropped morph if the receiving morph doesn't want it? The default behaviour is for it to do nothing, that is, to sit on top of the receiving morph, but without interacting with it. A more intuitive behavior is for the dropped morph to go back to its original position. This can be achieved by the receiver answering `true` to the message `repelsMorph:event:` when it doesn't want the dropped morph:

```
ReceiverMorph >> repelsMorph: aMorph event: anEvent
  ^ (self wantsDroppedMorph: aMorph event: anEvent) not
```

That's all we need as far as the receiver is concerned.

Create instances of `ReceiverMorph` and `EllipseMorph` in a playground:

```
ReceiverMorph new openInWorld;
    bounds: (100@100 corner: 200@200).
EllipseMorph new openInWorld.
```

Try to drag and drop the yellow `EllipseMorph` onto the receiver. It will be rejected and sent back to its initial position.

To change this behaviour, change the color of the ellipse morph to the color blue (by sending it the message `color: Color blue;` right after `new`). Blue morphs should be accepted by the `ReceiverMorph`.

Let's create a specific subclass of `Morph`, named `DroppedMorph`, so we can experiment a bit more. Let us define a new kind of morph called `DroppedMorph`.

```
Morph subclass: #DroppedMorph
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-Morphic'
```

```
DroppedMorph >> initialize
  super initialize.
  color := Color blue.
  self position: 250 @ 100
```

Now we can specify what the dropped morph should do when it is rejected by the receiver; here it will stay attached to the mouse pointer:

```
DroppedMorph >> rejectDropMorphEvent: anEvent
  | h |
  h := anEvent hand.
  WorldState addDeferredUIMessage: [ h grabMorph: self ].
  anEvent wasHandled: true
```

Sending the `hand` message to an event answers the *hand*, an instance of `HandMorph` that represents the mouse pointer and whatever it holds. Here we tell the `World` that the hand should grab `self`, the rejected morph.

Create two instances of `DroppedMorph` of different colors, and then drag and drop them onto the receiver.

```
ReceiverMorph new openInWorld.
morph := (DroppedMorph new color: Color blue) openInWorld.
morph position: (morph position + (70@0)).
(DroppedMorph new color: Color green) openInWorld.
```

The green morph is rejected and therefore stays attached to the mouse pointer.

15.11 A complete example

Let's design a morph to roll a die. Clicking on it will display the values of all sides of the die in a quick loop, and another click will stop the animation.

Define the die as a subclass of `BorderedMorph` instead of `Morph`, because we will make use of the border.

```
BorderedMorph subclass: #DieMorph
  instanceVariableNames: 'faces dieValue isStopped'
  classVariableNames: ''
  package: 'PBE-Morphic'
```

The instance variable `faces` records the number of faces on the die; we allow dice with up to 9 faces! `dieValue` records the value of the face that is currently displayed, and `isStopped` is true if the die animation has stopped running. To create a die instance, we define the `faces: n` method on the *class* side of `DieMorph` to create a new die with `n` faces.

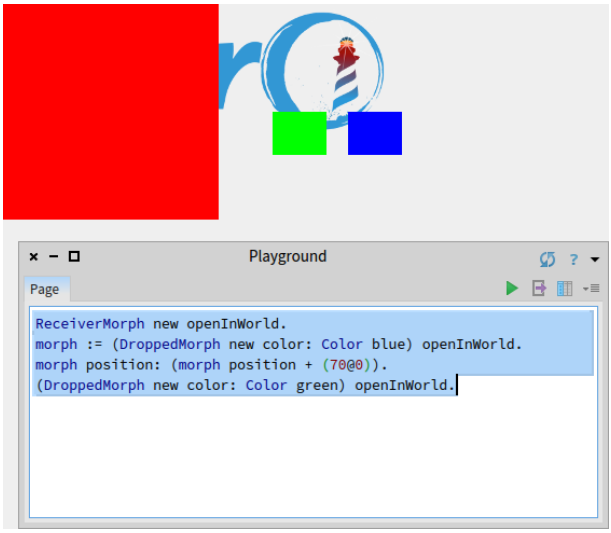

Figure 15-15 Creation of `DroppedMorph` and `ReceiverMorph`.

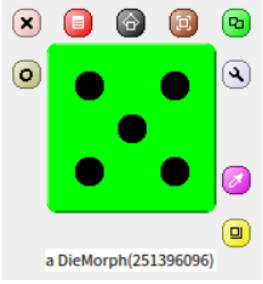

Figure 15-16 The die in Morphic

```
DieMorph class >> faces: aNumber
  ^ self new faces: aNumber
```

The `initialize` method is defined on the instance side in the usual way; remember that `new` automatically sends `initialize` to the newly-created instance.

```
DieMorph >> initialize
  super initialize.
  self extent: 50 @ 50.
  self
    useGradientFill;
    borderWidth: 2;
    useRoundedCorners.
  self setBorderStyle: #complexRaised.
```

```
  self fillStyle direction: self extent.
  self color: Color green.
  dieValue := 1.
  faces := 6.
  isStopped := false
```

We use a few methods of `BorderedMorph` to give a nice appearance to the die: a thick border with a raised effect, rounded corners, and a color gradient on the visible face. We define the instance method `faces:` to check for a valid parameter as follows:

```
DieMorph >> faces: aNumber
  "Set the number of faces"

  ((aNumber isInteger and: [ aNumber > 0 ]) and: [ aNumber <= 9 ])
    ifTrue: [ faces := aNumber ]
```

It may be good to review the order in which the messages are sent when a die is created. For instance, if we start by evaluating `DieMorph faces: 9`:

- The class method `DieMorph class >> faces:` sends `new` to `DieMorph class`.
- The method for `new` (inherited by `DieMorph class` from `Behavior`) creates the new instance and sends it the `initialize` message.
- The `initialize` method in `DieMorph` sets `faces` to an initial value of 6.
- `DieMorph class >> new` returns to the class method `DieMorph class >> faces:`, which then sends the message `faces: 9` to the new instance.
- The instance method `DieMorph >> faces:` now executes, setting the `faces` instance variable to 9.

Before defining `drawOn:`, we need a few methods to place the dots on the displayed face:

```
DieMorph >> face1
  ^ {(0.5 @ 0.5)}
```

```
DieMorph >> face2
  ^{0.25@0.25 . 0.75@0.75}
```

```
DieMorph >> face3
  ^{0.25@0.25 . 0.75@0.75 . 0.5@0.5}
```

```
DieMorph >> face4
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}
```

```
DieMorph >> face5
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}
```

Listing 15-17 Create a Die 6

```
(DieMorph faces: 6) openInWorld.
```

```
DieMorph >> face6
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5}
```

```
DieMorph >> face7
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5 . 0.5@0.5}
```

```
DieMorph >> face8
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5 . 0.5@0.5 . 0.5@0.25}
```

```
DieMorph >> face9
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5 . 0.5@0.5 . 0.5@0.25 . 0.5@0.75}
```

These methods define collections of the coordinates of dots for each face. The coordinates are in a square of size 1x1; we will simply need to scale them to place the actual dots.

The `drawOn:` method does two things: it draws the die background with the `super`-send, and then draws the dots as follows:

```
DieMorph >> drawOn: aCanvas
  super drawOn: aCanvas.
  (self perform: ('face', dieValue asString) asSymbol)
    do: [:aPoint | self drawDotOn: aCanvas at: aPoint]
```

The second part of this method uses the reflective capacities of Pharo. Drawing the dots of a face is a simple matter of iterating over the collection given by the `faceX` method for that face, sending the `drawDotOn:at:` message for each coordinate. To call the correct `faceX` method, we use the `perform:` method which sends a message built from a string, `('face', dieValue asString) asSymbol`.

```
DieMorph >> drawDotOn: aCanvas at: aPoint
  aCanvas
    fillOval: (Rectangle
      center: self position + (self extent * aPoint)
      extent: self extent / 6)
    color: Color black
```

Since the coordinates are normalized to the `[0:1]` interval, we scale them to the dimensions of our die: `self extent * aPoint`. We can already create a die instance from a playground (see result on Figure 15-18):

To change the displayed face, we create an accessor that we can use as `myDie dieValue: 5`:

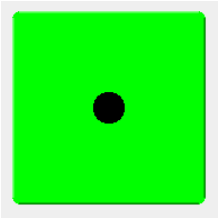

Figure 15-18 A new die 6 with `(DieMorph faces: 6) openInWorld`

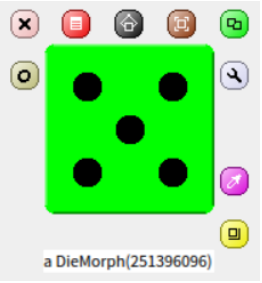

Figure 15-19 Result of `(DieMorph faces: 6) openInWorld; dieValue: 5`.

```
DieMorph >> dieValue: aNumber
  ((aNumber isInteger and: [ aNumber > 0 ]) and: [ aNumber <= faces
    ])
    ifTrue: [
      dieValue := aNumber.
      self changed ]
```

Now we will use the animation system to show quickly all the faces:

```
DieMorph >> stepTime
  ^ 100
```

```
DieMorph >> step
  isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]
```

Now the die is rolling!

To start or stop the animation by clicking, we will use what we learned previously about mouse events. First, activate the reception of mouse events:

```
DieMorph >> handlesMouseDown: anEvent
  ^ true
```

Second, we will stop and start alternatively a roll on mouse click.

Figure 15-20 The die displayed with alpha-transparency

```
DieMorph >> mouseDown: anEvent
  anEvent redButtonPressed
    ifTrue: [isStopped := isStopped not]
```

Now the die will roll or stop rolling when we click on it.

15.12 More about the canvas

The `drawOn:` method has an instance of `Canvas` as its sole argument; the canvas is the area on which the morph draws itself. By using the graphics methods of the canvas you are free to give the appearance you want to a morph. If you browse the inheritance hierarchy of the `Canvas` class, you will see that it has several variants. The default variant of `Canvas` is `FormCanvas`, and you will find the key graphics methods in `Canvas` and `FormCanvas`. These methods can draw points, lines, polygons, rectangles, ellipses, text, and images with rotation and scaling.

It is also possible to use other kinds of canvas, for example to obtain transparent morphs, more graphics methods, antialiasing, and so on. To use these features you will need an `AlphaBlendingCanvas` or a `BalloonCanvas`. But how can you obtain such a canvas in a `drawOn:` method, when `drawOn:` receives an instance of `FormCanvas` as its argument? Fortunately, you can transform one kind of canvas into another.

To use a canvas with a 0.5 alpha-transparency in `DieMorph`, redefine `drawOn:` like this:

```
DieMorph >> drawOn: aCanvas
  | theCanvas |
  theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
  super drawOn: theCanvas.
  (self perform: ('face', dieValue asString) asSymbol)
    do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

That's all you need to do!

15.13 Chapter summary

Morphic is a graphical framework in which graphical interface elements can be dynamically composed.

- You can convert an object into a morph and display that morph on the screen by sending it the messages `asMorph openInWorld`.
- You can manipulate a morph by meta-clicking on it and using the handles that appear. (Handles have help balloons that explain what they do.)
- You can compose morphs by embedding one onto another, either by drag and drop or by sending the message `addMorph:`.
- You can subclass an existing morph class and redefine key methods, like `initialize` and `drawOn:`.
- You can control how a morph reacts to mouse and keyboard events by redefining the methods `handlesMouseDown:`, `handlesMouseOver:`, etc.
- You can animate a morph by defining the methods `step` (what to do) and `stepTime` (the number of milliseconds between steps).

CHAPTER 16

Classes and metaclasses

As we saw in preceding chapters, in Pharo, everything is an object, and every object is an instance of a class. Classes are no exception: classes are objects, and class objects are instances of other classes. This object model captures the essence of object-oriented programming, and is lean, simple, elegant and uniform. However, the implications of this uniformity may confuse newcomers.

Note that you do not need to fully understand the implications of this uniformity to program fluently in Pharo. Nevertheless, the goal of this chapter is twofold: (1) go as deep as possible and (2) show that there is nothing complex, *magic* or special here: just simple rules applied uniformly. By following these rules you can always understand why the situation is the way that it is.

16.1 Rules for classes

The Pharo object model is based on a limited number of concepts applied uniformly. To refresh your memory, here are the rules of the object model that we explored in Chapter : The Pharo Object Model.

Rule 1 Everything is an object.

Rule 2 Every object is an instance of a class.

Rule 3 Every class has a superclass.

Rule 4 Everything happens by sending messages.

Rule 5 Method lookup follows the inheritance chain.

Rule 6 Classes are objects too and follow exactly the same rules.

As we mentioned in the introduction to this chapter, a consequence of Rule 1 is that *classes are objects too*, so Rule 2 tells us that classes must also be instances of classes. The class of a class is called a *metaclass*.

16.2 Metaclasses

A metaclass is created automatically for you whenever you create a class. Most of the time you do not need to care or think about metaclasses. However, every time that you use the browser to browse the *class side* of a class, it is helpful to recall that you are actually browsing a different class. A class and its metaclass are two separate classes, even though the former is an instance of the latter.

To properly explain classes and metaclasses, we need to extend the rules from Chapter : The Pharo Object Model with the following additional rules.

Rule 7 Every class is an instance of a metaclass.

Rule 8 The metaclass hierarchy parallels the class hierarchy.

Rule 9 Every metaclass inherits from `Class` and `Behavior`.

Rule 10 Every metaclass is an instance of `Metaclass`.

Rule 11 The metaclass of `Metaclass` is an instance of `Metaclass`.

Together, these 11 rules complete Pharo's object model.

We will first briefly revisit the 5 rules from Chapter : The Pharo Object Model with a small example. Then we will take a closer look at the new rules, using the same example.

16.3 Revisiting the Pharo object model

Rule 1. Since everything is an object, an ordered collection in Pharo is also an object.

```
OrderedCollection withAll: #(4 5 6 1 2 3)
>>> an OrderedCollection(4 5 6 1 2 3)
```

Rule 2. Every object is an instance of a class. The class of an ordered collection is the class `OrderedCollection`:

```
(OrderedCollection withAll: #(4 5 6 1 2 3)) class
>>> OrderedCollection
```

Rule 3. Every class has a superclass. The superclass of `OrderedCollection` is `SequenceableCollection` and the superclass of `SequenceableCollection` is `Collection`:

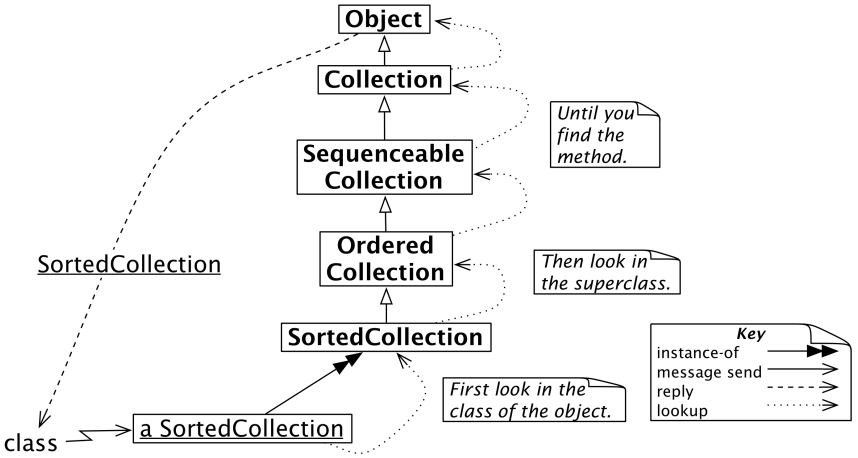

Figure 16-1 Sending the message `class` to a sorted collection

```
OrderedCollection superclass
>>> SequenceableCollection
```

```
SequenceableCollection superclass
>>> Collection
```

```
Collection superclass
>>> Object
```

Let us take an example. When we send the message `asSortedCollection`, we convert the ordered collection into a sorted collection. We verify simply as follows:

```
(OrderedCollection withAll: #(4 5 6 1 2 3)) asSortedCollection class
>>> SortedCollection
```

Rule 4. Everything happens by sending messages, so we can deduce that `withAll:` is a message to `OrderedCollection` and `asSortedCollection` are messages sent to the ordered collection instance, and `superclass` is a message to `OrderedCollection` and `SequenceableCollection`, and `Collection`. The receiver in each case is an object, since everything is an object, but some of these objects are also classes.

Rule 5. Method lookup follows the inheritance chain, so when we send the message `class` to the result of `(OrderedCollection withAll: #(4 5 6 1 2 3)) asSortedCollection`, the message is handled when the corresponding method is found in the class `Object`, as shown in Figure 16-1.

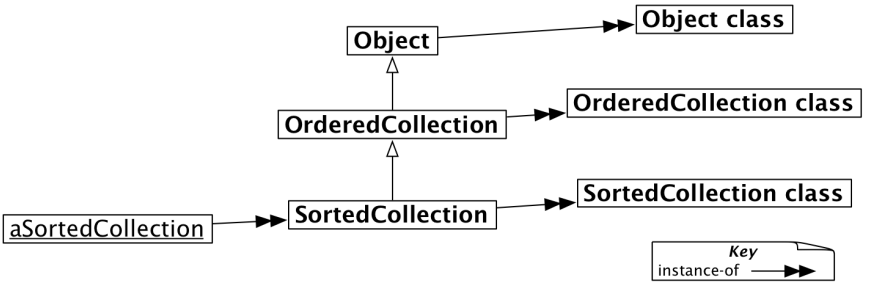

Figure 16-2 The metaclasses of `SortedCollection` and its superclasses (elided).

16.4 Every class is an instance of a metaclass

As we mentioned earlier in Section 16.2, classes whose instances are themselves classes are called metaclasses.

Metaclasses are implicit

Metaclasses are automatically created when you define a class. We say that they are *implicit* since as a programmer you never have to worry about them. An implicit metaclass is created for each class you create, so each metaclass has only a single instance.

Whereas ordinary classes are named, metaclasses are anonymous. However, we can always refer to them through the class that is their instance. The class of `SortedCollection`, for instance, is `SortedCollection class`, and the class of `Object` is `Object class`:

```
SortedCollection class
>>> SortedCollection class
```

```
Object class
>>> Object class
```

In fact metaclasses are not truly anonymous, their name is deduced from the one of their single instance.

```
SortedCollection class name
>>> 'SortedCollection class'
```

Figure 16-2 shows how each class is an instance of its metaclass. Note that we only skip `SequenceableCollection` and `Collection` from the figures and explanation due to space constraints. Their absence does not change the overall meaning.

16.5 Querying Metaclasses

The fact that classes are also objects makes it easy for us to query them by sending messages. Let's have a look:

```
OrderedCollection subclasses
>>> {SortedCollection . ObjectFinalizerCollection .
    WeakOrderedCollection . OCLiteralList . GLMMultiValue}
```

```
SortedCollection subclasses
>>> #()
```

```
SortedCollection allSuperclasses
>>> an OrderedCollection(OrderedCollection SequenceableCollection
    Collection Object ProtoObject)
```

```
SortedCollection instVarNames
>>> #(#sortBlock)
```

```
SortedCollection allInstVarNames
>>> #(#array #firstIndex #lastIndex #sortBlock)
```

```
SortedCollection selectors
>>>  #(#sortBlock: #add: #groupedBy: #defaultSort:to: #addAll:
    #at:put: #copyEmpty #, #collect: #indexForInserting:
    #insert:before: #reSort #addFirst: #join: #median #flatCollect:
    #sort: #sort:to: #= #sortBlock)
```

16.6 The metaclass hierarchy parallels the class hierarchy

Rule 7 says that the superclass of a metaclass cannot be an arbitrary class: it is constrained to be the metaclass of the superclass of the metaclass's unique instance.

```
SortedCollection class superclass
>>> OrderedCollection class
```

```
SortedCollection superclass class
>>> OrderedCollection class
```

This is what we mean by the metaclass hierarchy being parallel to the class hierarchy. Figure 16-3 shows how this works in the `SortedCollection` hierarchy.

```
SortedCollection class
>>> SortedCollection class
```

```
SortedCollection class superclass
>>> OrderedCollection class
```

```
SortedCollection class superclass superclass
>>> SequenceableCollection class
```

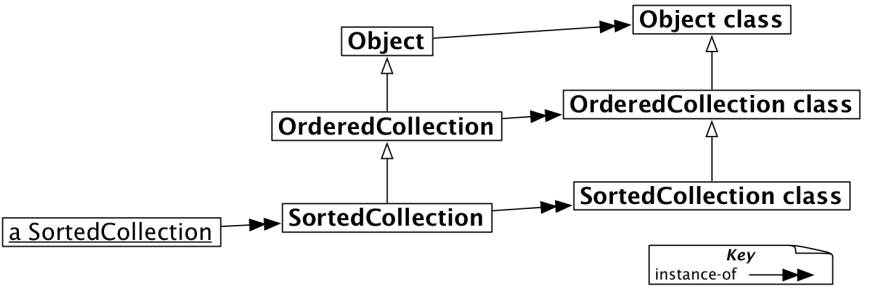

Figure 16-3 The metaclass hierarchy parallels the class hierarchy (elided).

```
SortedCollection class superclass superclass superclass superclass
>>> Object class
```

Uniformity between Classes and Objects

It is interesting to step back a moment and realize that there is no difference between sending a message to an object and to a class. In both cases the search for the corresponding method starts in the *class of the receiver*, and *proceeds up the inheritance chain.*

Thus, messages sent to classes must follow the metaclass inheritance chain. Consider, for example, the method `withAll:`, which is implemented on the class side of `Collection`. When we send the message `withAll:` to the class `OrderedCollection`, then it is looked up the same way as any other message. The lookup starts in `OrderedCollection class` (since it starts in the class of the receiver and the receiver is `OrderedCollection`), and proceeds up the metaclass hierarchy until it is found in `Collection class` (see Figure 16-4). It returns a new instance of `OrderedCollection`.

```
OrderedCollection withAll: #(4 5 6 1 2 3)
>>> an OrderedCollection (4 5 6 1 2 3)
```

Only one method lookup

Thus we see that there is one uniform kind of method lookup in Pharo. Classes are just objects, and behave like any other objects. Classes have the power to create new instances only because classes happen to respond to the message `new`, and because the method for `new` knows how to create new instances. Normally, non-class objects do not understand this message, but if you have a good reason to do so, there is nothing stopping you from adding a `new` method to a non-metaclass.

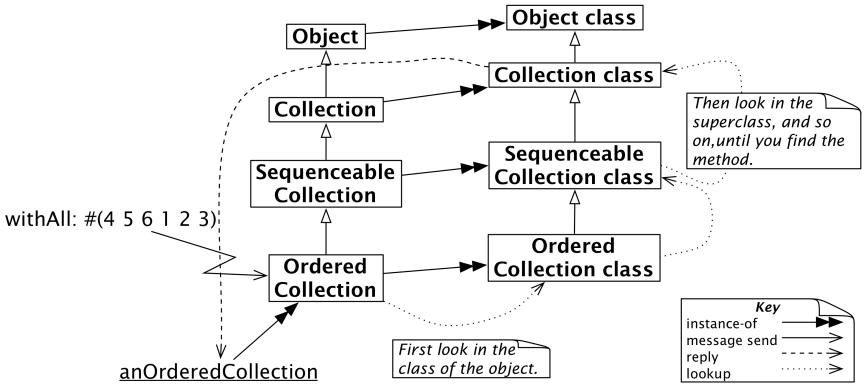

Figure 16-4 Message lookup for classes is the same as for ordinary objects.

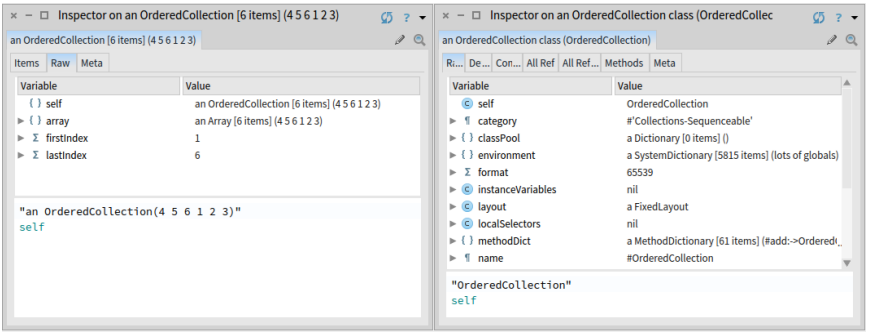

Figure 16-5 Classes are objects too.

Inspecting objects and classes

Since classes are objects, we can also inspect them.

Inspect `OrderedCollection withAll: #(4 5 6 1 2 3)` and `OrderedCollection`.

Notice that in one case you are inspecting an instance of `OrderedCollection` and in the other case the `OrderedCollection` class itself. This can be a bit confusing, because the title bar of the inspector names the *class* of the object being inspected.

The inspector on `OrderedCollection` allows you to see the superclass, instance variables, method dictionary, and so on, of the `OrderedCollection` class, as shown in Figure 16-5.

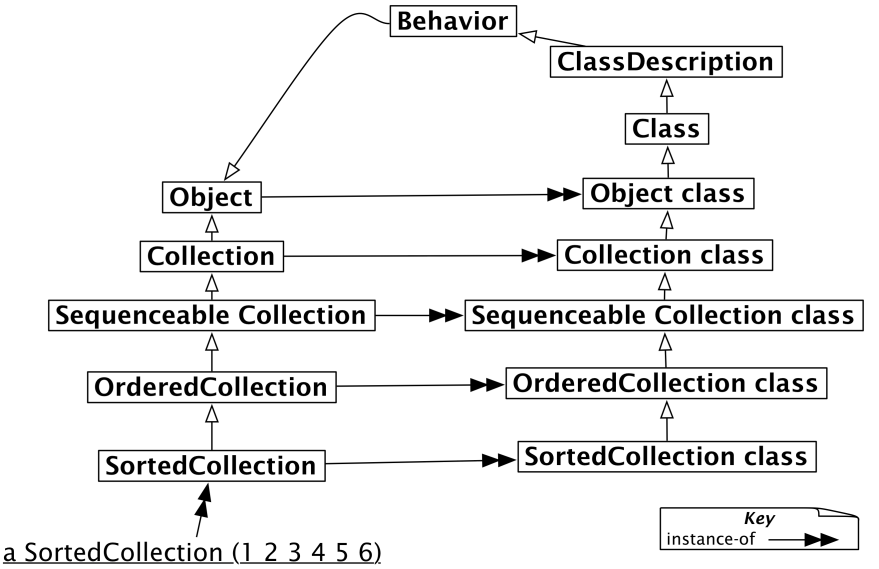

Figure 16-6 Metaclasses inherit from `Class` and `Behavior`.

16.7 Every metaclass inherits from `Class` and `Behavior`

Every metaclass is a kind of a class (a class with a single instance), hence inherits from `Class`. `Class` in turn inherits from its superclasses, `ClassDescription` and `Behavior`. Since everything in Pharo is an object, these classes all inherit eventually from `Object`. We can see the complete picture in Figure 16-6.

Where is `new` defined?

To understand the importance of the fact that metaclasses inherit from `Class` and `Behavior`, it helps to ask where `new` is defined and how it is found. When the message `new` is sent to a class, it is looked up in its metaclass chain and ultimately in its superclasses `Class`, `ClassDescription` and `Behavior` as shown in Figure 16-7.

The question *Where is* `new` *defined?* is crucial. `new` is first defined in the class `Behavior`, and it can be redefined in its subclasses, including any of the metaclass of the classes we define, when this is necessary. Now when a message `new` is sent to a class it is looked up, as usual, in the metaclass of this class, continuing up the superclass chain right up to the class `Behavior`, if it has not been redefined along the way.

Note that the result of sending `SortedCollection new` is an instance of `SortedCollection` and *not* of `Behavior`, even though the method is looked

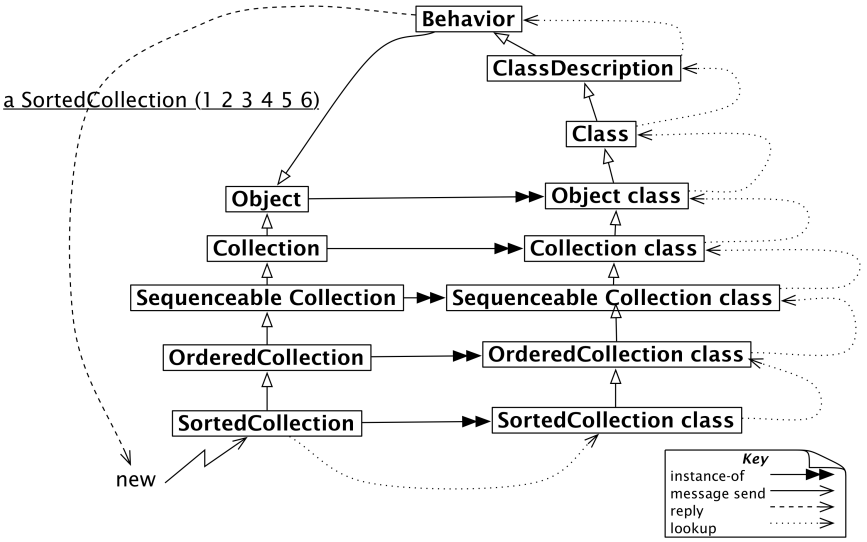

Figure 16-7 new is an ordinary message looked up in the metaclass chain.

up in the class Behavior! new always returns an instance of self, the class that receives the message, even if it is implemented in another class.

```
SortedCollection new class
>>> SortedCollection    "not Behavior!"
```

Common mistake.

A common mistake is to look for new in the superclass of the receiving class. The same holds for new:, the standard message to create an object of a given size. For example, Array new: 4 creates an array of 4 elements. You will not find this method defined in Array or any of its superclasses. Instead you should look in Array class and its superclasses, since that is where the lookup will start (See Figure 16-7).

Responsibilities of Behavior, ClassDescription, and Class

Behavior provides the minimum state necessary for objects that have instances, which includes a superclass link, a method dictionary and the class format. The class format is an integer that encodes the pointer/non-pointer distinction, compact/non-compact class distinction, and basic size of instances. Behavior inherits from Object, so it, and all of its subclasses, can behave like objects.

Behavior is also the basic interface to the compiler. It provides methods for creating a method dictionary, compiling methods, creating instances

(*i.e.*, `new`, `basicNew`, `new:`, and `basicNew:`), manipulating the class hierarchy (*i.e.*, `superclass:`, `addSubclass:`), accessing methods (*i.e.*, `selectors`, `allSelectors`, `compiledMethodAt:`), accessing instances and variables (*i.e.*, `allInstances`, `instVarNames`...), accessing the class hierarchy (*i.e.*, `superclass`, `subclasses`) and querying (*i.e.*, `hasMethods`, `includesSelector`, `canUnderstand:`, `inheritsFrom:`, `isVariable`).

`ClassDescription` is an abstract class that provides facilities needed by its two direct subclasses, `Class` and `Metaclass`. `ClassDescription` adds a number of facilities to the base provided by `Behavior`: named instance variables, the categorization of methods into protocols, the maintenance of change sets and the logging of changes, and most of the mechanisms needed for filing out changes.

`Class` represents the common behaviour of all classes. It provides a class name, compilation methods, method storage, and instance variables. It provides a concrete representation for class variable names and shared pool variables (`addClassVarName:`, `addSharedPool:`, `initialize`). Since a metaclass is a class for its sole instance (*i.e.*, the non-meta class), all metaclasses ultimately inherit from `Class` (as shown by Figure 16-9).

16.8 Every metaclass is an instance of `Metaclass`

One question left is since metaclasses are objects too, they should be instances of another class, but which one? Metaclasses are objects too; they are instances of the class `Metaclass` as shown in Figure 16-8. The instances of class `Metaclass` are the anonymous metaclasses, each of which has exactly one instance, which is a class.

`Metaclass` represents common metaclass behaviour. It provides methods for instance creation (`subclassOf:`), creating initialized instances of the metaclass's sole instance, initialization of class variables, metaclass instance, method compilation, and class information (inheritance links, instance variables, etc.).

16.9 The metaclass of `Metaclass` is an instance of `Metaclass`

The final question to be answered is: what is the class of `Metaclass class`? The answer is simple: it is a metaclass, so it must be an instance of `Metaclass`, just like all the other metaclasses in the system (see Figure 16-9).

Figure 16-9 shows how all metaclasses are instances of `Metaclass`, including the metaclass of `Metaclass` itself. If you compare Figures 16-8 and 16-9 you will see how the metaclass hierarchy perfectly mirrors the class hierarchy, all the way up to `Object class`.

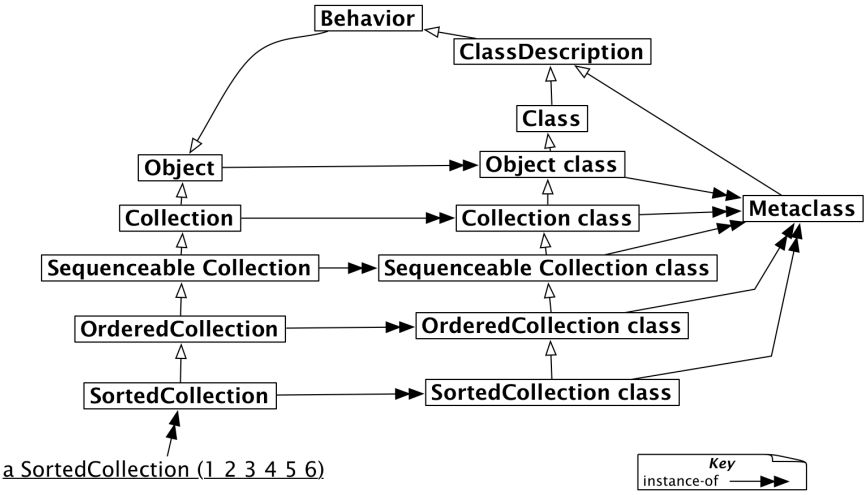

Figure 16-8 Every metaclass is a `Metaclass`.

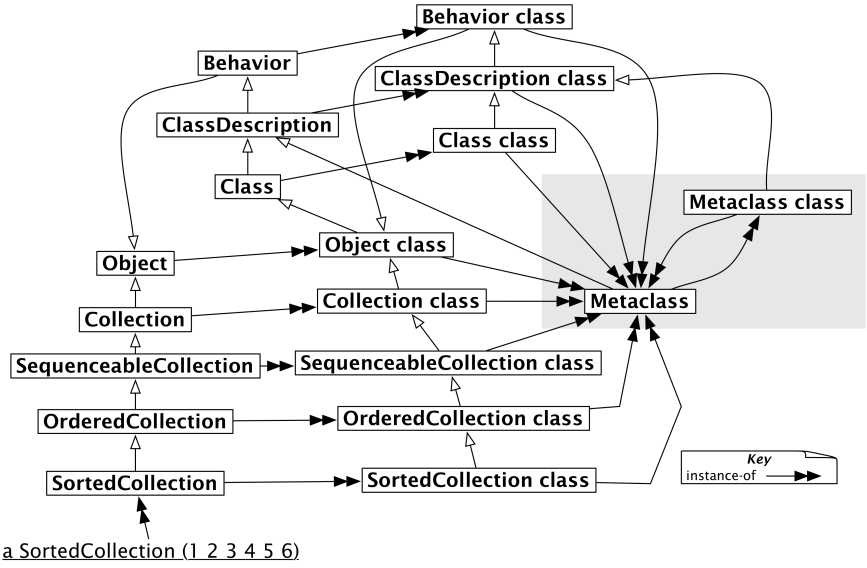

Figure 16-9 All metaclasses are instances of the class `Metaclass`, even the metaclass of `Metaclass`.

Listing 16-10 The class hierarchy

```
Collection superclass
>>> Object
```

Listing 16-11 The parallel metaclass hierarchy

```
Collection class superclass
>>> Object class
[[[testcase=true
Object class superclass superclass
>>> Class    "NB: skip ProtoObject class"
```

Listing 16-12 Instances of Metaclass

```
Collection class class
>>> Metaclass
```

The following examples show us how we can query the class hierarchy to demonstrate that Figure 16-9 is correct. (Actually, you will see that we told a white lie — `Object class superclass --> ProtoObject class`, not `Class`. In Pharo, we must go one superclass higher to reach `Class`.)

```
Class superclass
>>> ClassDescription
```

```
ClassDescription superclass
>>> Behavior
```

```
Behavior superclass
>>> Object
```

```
Object class class
>>> Metaclass
```

```
Behavior class class
>>> Metaclass
```

```
Metaclass superclass
>>> ClassDescription
```

16.10 Chapter summary

This chapter gave an in-depth look into the uniform object model, and a more thorough explanation of how classes are organized. If you get lost or confused, you should always remember that message passing is the key: you look for the method in the class of the receiver. This works on *any* receiver.

Listing 16-13 Metaclass class is a Metaclass

```
Metaclass class class
>>> Metaclass
```

If the method is not found in the class of the receiver, it is looked up in its superclasses.

- Every class is an instance of a metaclass. Metaclasses are implicit. A metaclass is created automatically when you create the class that is its sole instance. A metaclass is simply a class whose unique instance is a class.
- The metaclass hierarchy parallels the class hierarchy. Method lookup for classes parallels method lookup for ordinary objects, and follows the metaclass's superclass chain.
- Every metaclass inherits from `Class` and `Behavior`. Every class *is a* `Class`. Since metaclasses are classes too, they must also inherit from `Class`. `Behavior` provides behavior common to all entities that have instances.
- Every metaclass is an instance of `Metaclass`. `ClassDescription` provides everything that is common to `Class` and `Metaclass`.
- The metaclass of `Metaclass` is an instance of `Metaclass`. The *instance-of* relation forms a closed loop, so `Metaclass class class` is `Metaclass`.

CHAPTER 17

Reflection

Pharo is a reflective programming language. In a nutshell, this means that programs are able to *reflect* on their own execution and structure. More technically, this means that the *metaobjects* of the runtime system can be *reified* as ordinary objects, which can be queried and inspected. The metaobjects in Pharo are classes, metaclasses, method dictionaries, compiled methods, but also the run-time stack, processes, and so on. This form of reflection is also called *introspection*, and is supported by many modern programming languages.

Conversely, it is possible in Pharo to modify reified metaobjects and *reflect* these changes back to the runtime system (see Figure 17-1). This is also called *intercession*, and is supported mainly by dynamic programming languages,

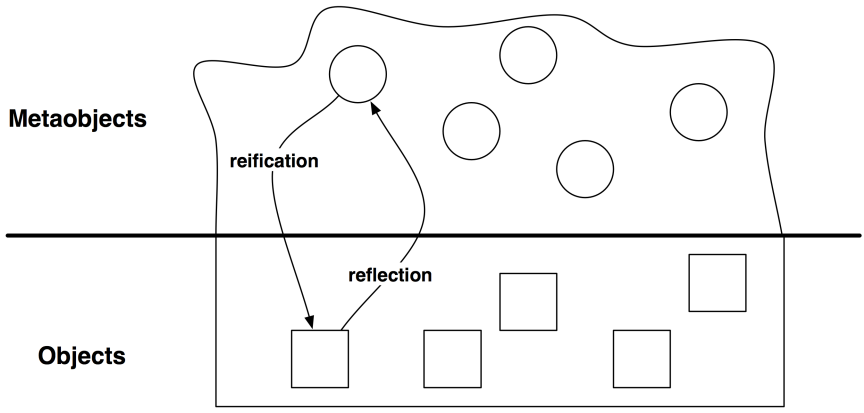

Figure 17-1 Reification and reflection.

and only to a very limited degree by static languages. So pay attention when people say that Java is a reflective language, it is an introspective one not a reflective one.

A program that manipulates other programs (or even itself) is a *metaprogram*. For a programming language to be reflective, it should support both introspection and intercession. Introspection is the ability to *examine* the data structures that define the language, such as objects, classes, methods and the execution stack. Intercession is the ability to *modify* these structures, in other words to change the language semantics and the behavior of a program from within the program itself. *Structural reflection* is about examining and modifying the structures of the run-time system, and *behavioural reflection* is about modifying the interpretation of these structures.

In this chapter we will focus mainly on structural reflection. We will explore many practical examples illustrating how Pharo supports introspection and metaprogramming.

17.1 Introspection

Using the inspector, you can look at an object, change the values of its instance variables, and even send messages to it.

Evaluate the following code in a playground:

```
w := GTPlayground openLabel: 'My Playground'.
w inspect
```

This will open a second playground and an inspector. The inspector shows the internal state of this new playground, listing its instance variables on the left (`borderColor`, `borderWidth`, `bounds`...) and the value of the selected instance variable on the right. The `bounds` instance variable represents the precise area occupied by the playground.

Now choose the inspector and click the playground area of the inspector which has a comment on top and type `self bounds: (Rectangle origin: 10@10 corner: 300@300 )` in it as shown in Figure 17-2 and then *Do It* like you do with a code of a Playground.

Immediately you will see the Playground that we created will change and resize itself.

Accessing instance variables

How does the inspector work? In Pharo, all instance variables are protected. In theory, it is impossible to access them from another object if the class doesn't define any accessor. In practice, the inspector can access instance variables without needing accessors, because it uses the reflective abilities of Pharo. Classes define instance variables either by name or by numeric

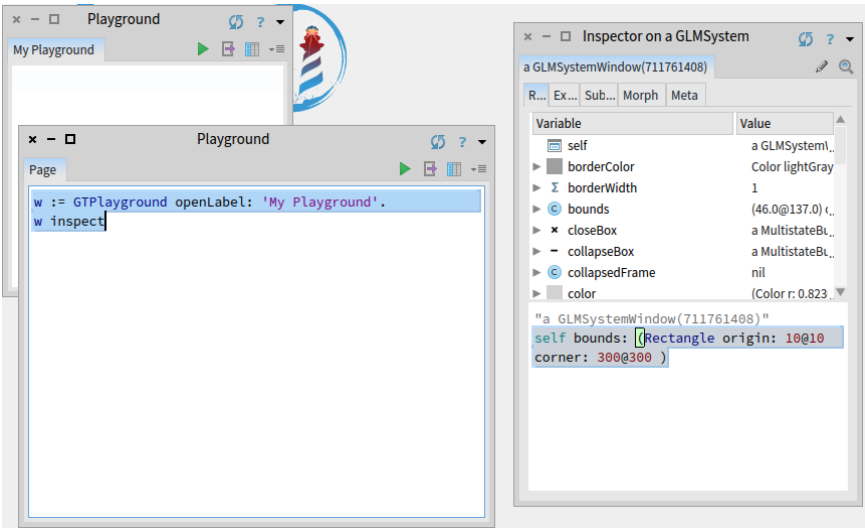

Figure 17-2 Inspecting a `Workspace`.

indices. The inspector uses methods defined by the `Object` class to access them: `instVarAt: index` and `instVarNamed: aString` can be used to get the value of the instance variable at position `index` or identified by `aString`, respectively. Similarly, to assign new values to these instance variables, it uses `instVarAt:put:` and `instVarNamed:put:`.

For instance, you can change the value of the `w` binding of the first workspace by evaluating:

```
w instVarNamed:'bounds' put: (Rectangle origin: 10@10 corner:
    500@500).
```

> **Important** *Caveat: Although these methods are useful for building development tools, using them to develop conventional applications is a bad idea: these reflective methods break the encapsulation boundary of your objects and can therefore make your code much harder to understand and maintain.*

Both `instVarAt:` and `instVarAt:put:` are primitive methods, meaning that they are implemented as primitive operations of the Pharo virtual machine. If you consult the code of these methods, you will see the special pragma syntax `<primitive: N>` where `N` is an integer.

```
Object >> instVarAt: index
  "Primitive. Answer a fixed variable in an object. ..."

  <primitive: 173 error: ec>
  self primitiveFailed
```

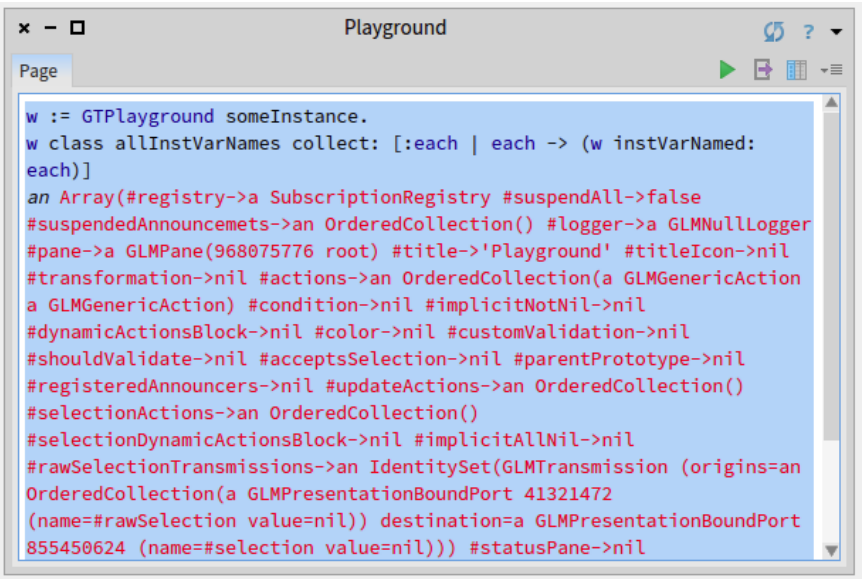

Figure 17-3 Displaying all instance variables of a `GTPlayground`.

Any Pharo code after the primitive declaration is executed only if the primitive fails. This also allows the debugger to be started on primitive methods. In this specific case, there is no way to implement this method, so the whole method just fails.

Other methods are implemented on the VM for faster execution. For example some arithmetic operations on `SmallInteger`:

```
* aNumber
  "Primitive. Multiply the receiver by the argument and answer with
    the
  result if it is a SmallInteger. Fail if the argument or the result
    is not a
  SmallInteger. Essential. No Lookup. See Object documentation
    whatIsAPrimitive."

  <primitive: 9>
  ^ super * aNumber
```

If this primitive fails, for example if the VM does not handle the type of the argument, the Pharo code is executed. Although it is possible to modify the code of primitive methods, beware that this can be risky business for the stability of your Pharo system.

Figure 17-3 shows how to display the values of the instance variables of an arbitrary instance (`w`) of class `GTPlayground`. The method `allInstVarNames`

returns all the names of the instance variables of a given class.

```
GTPlayground allInstVarNames
>>>
#(#registry #suspendAll #suspendedAnnouncemets #logger #pane #title
    #titleIcon #transformation #actions #condition #implicitNotNil
    #dynamicActionsBlock #color #customValidation #shouldValidate
    #acceptsSelection #parentPrototype #registeredAnnouncers
    #updateActions #selectionActions #selectionDynamicActionsBlock
    #implicitAllNil #rawSelectionTransmissions #statusPane
    #sourceLink #initializationBlock #cachedDisplayedValue
    #labelActionBlock #portChangeActions #wantsSteps #stepTime
    #stepCondition #wantsAutomaticRefresh #presentations
    #arrangement)
```

```
w := GTPlayground someInstance.
w class allInstVarNames collect: [:each | each -> (w instVarNamed:
    each)]
```

In the same spirit, it is possible to gather instances that have specific properties iterating over instances of a class using an iterator such as `select:`. For instance, to get all objects who are directly included in the world morph (the main root of the graphical displayed elements), try this expression:

```
Morph allSubInstances
  select: [ :each |
        | own |
        own := (each instVarNamed: 'owner').
        own isNotNil and: [ own  isWorldMorph ]]
>>>
OrderedCollection(a GLMSystemWindow(874014976) named: Playground a
    GLMSystemWindow(365303808) named: Playground a
    MenubarMorph(779936512) a HandMorph(725291008) a
    TaskbarMorph(747120896))
```

Querying classes and interfaces

The development tools in Pharo (system browser, debugger, inspector...) all use the reflective features we have seen so far.

Here are a few other messages that might be useful to build development tools:

`isKindOf: aClass` returns true if the receiver is instance of `aClass` or of one of its subclasses. For instance:

```
1.5 class
>>> SmallFloat64
```

```
1.5 isKindOf: Float
>>> true
```

```
1.5 isKindOf: Number
>>> true
```

```
1.5 isKindOf: Integer
>>> false
```

`respondsTo: aSymbol` returns true if the receiver has a method whose selector is `aSymbol`. For instance:

```
1.5 respondsTo: #floor
>>> true    "since Number implements floor"
```

```
1.5 floor
>>> 1
```

```
Exception respondsTo: #,
>>> true    "exception classes can be grouped"
```

Important Caveat:

Although these features are especially useful for implementing development tools, they are normally not appropriate for typical applications. Asking an object for its class, or querying it to discover which messages it understands, are typical signs of design problems, since they violate the principle of encapsulation. Development tools, however, are not normal applications, since their domain is that of software itself. As such these tools have a right to dig deep into the internal details of code.

Code metrics

Let's see how we can use Pharo's introspection features to quickly extract some code metrics. Code metrics measure aspects such as the depth of the inheritance hierarchy, the number of direct or indirect subclasses, the number of methods or instance variables in each class, or the number of locally defined methods or instance variables. Here are a few metrics for the class `Morph`, which is the superclass of all graphical objects in Pharo, revealing that it is a huge class, and that it is at the root of a huge hierarchy. Maybe it needs some refactoring!

```
"inheritance depth"
Morph allSuperclasses size.
>>> 2
```

```
"number of methods"
Morph allSelectors size.
>>> 1346
```

```
"number of instance variables"
Morph allInstVarNames size.
>>> 6
```

```
"number of new methods"
Morph selectors size.
>>> 905
```

```
"number of new variables"
Morph instVarNames size.
>>> 6
```

```
"direct subclasses"
Morph subclasses size.
>>> 65
```

```
"total subclasses"
Morph allSubclasses size.
>>>  428
```

```
"total lines of code!"
Morph linesOfCode.
>>> 5027
```

One of the most interesting metrics in the domain of object-oriented languages is the number of methods that extend methods inherited from the superclass. This informs us about the relation between the class and its superclasses. In the next sections we will see how to exploit our knowledge of the runtime structure to answer such questions.

17.2 Browsing code

In Pharo, everything is an object. In particular, classes are objects that provide useful features for navigating through their instances. Most of the messages we will look at now are implemented in `Behavior`, so they are understood by all classes.

For example, you can obtain a random instance of a given class by sending it the message `someInstance`.

```
Point someInstance
>>> (-1@-1)
```

You can also gather all the instances with `allInstances`, or the number of active instances in memory with `instanceCount`.

```
ByteString allInstances
>>> #('collection' 'position'  ...)
```

```
ByteString instanceCount
>>> 58514
```

```
String allSubInstances size
>>>  138962
```

These features can be very useful when debugging an application, because you can ask a class to enumerate those of its methods exhibiting specific properties. Here are some more interesting and useful methods for code discovery through reflection.

`whichSelectorsAccess:` returns the list of all selectors of methods that read or write the instance variable named by the argument

`whichSelectorsStoreInto:` returns the selectors of methods that modify the value of an instance variable

`whichSelectorsReferTo:` returns the selectors of methods that send a given message

```
Point whichSelectorsAccess: 'x'
>>> #(#octantOf: #roundDownTo: #+ #asIntegerPoint #transposed ...)
```

```
Point whichSelectorsStoreInto: 'x'
>>> #(#fromSton: #setX:setY: #setR:degrees: #bitShiftPoint:)
```

```
Point whichSelectorsReferTo: #+
>>> #(#+)
```

The following messages take inheritance into account:

`whichClassIncludesSelector:` returns the superclass that implements the given message

`unreferencedInstanceVariables` returns the list of instance variables that are neither used in the receiver class nor any of its subclasses

```
Rectangle whichClassIncludesSelector: #inspect
>>> Object
```

```
Rectangle unreferencedInstanceVariables
>>> #()
```

`SystemNavigation` is a facade that supports various useful methods for querying and browsing the source code of the system. `SystemNavigation default` returns an instance you can use to navigate the system. For example:

```
SystemNavigation default allClassesImplementing: #yourself
>>> an OrderedCollection(Object)
```

The following messages should also be self-explanatory:

```
SystemNavigation default allSentMessages size
>>>43985
```

```
(SystemNavigation default allUnsentMessagesIn: Object selectors) size
>>> 37
```

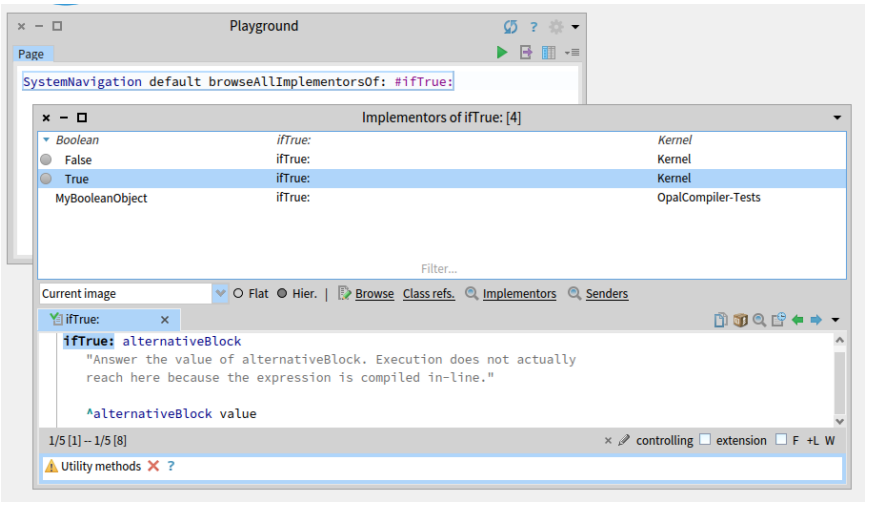

Figure 17-4 Browse all implementations of `ifTrue:`.

```
SystemNavigation default allUnimplementedCalls size
>>> 269
```

Note that messages implemented but not sent are not necessarily useless, since they may be sent implicitly (*e.g.*, using `perform:`). Messages sent but not implemented, however, are more problematic, because the methods sending these messages will fail at runtime. They may be a sign of unfinished implementation, obsolete APIs, or missing libraries.

`Point allCallsOn` returns all messages sent explicitly to `Point` as a receiver.

All these features are integrated into the programming environment of Pharo, in particular the code browsers. As we mentioned before, there are convenient keyboard shortcuts for browsing all implementors (`CMD-m`) and browsing senders (`CMD-n`) of a given message. What is perhaps not so well known is that there are many such pre-packaged queries implemented as methods of the `SystemNavigation` class in the `query` protocol. For example, you can programmatically browse all implementors of the message `ifTrue:` by evaluating:

```
SystemNavigation default browseAllImplementorsOf: #ifTrue:
```

Particularly useful are the methods `browseAllSelect:` and `browseMethodsWithSourceString:matchCase:`. Here are two different ways to browse all methods in the system that perform super sends (the first way is rather brute force, the second way is better and eliminates some false positives):

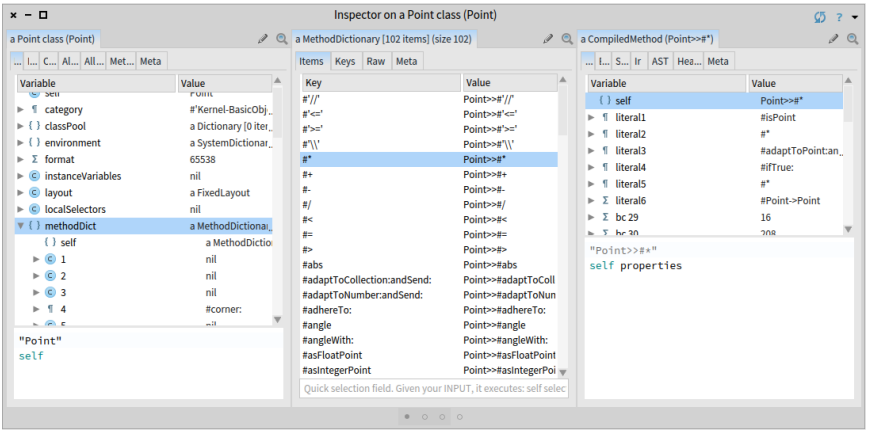

Figure 17-5 Inspector on class `Point` and the bytecode of its `#*` method.

```
SystemNavigation default browseMethodsWithSourceString: 'super'
    matchCase: true.
SystemNavigation default browseAllSelect: [:method | method
    sendsToSuper ].
```

17.3 Classes, method dictionaries and methods

Since classes are objects, we can inspect or explore them just like any other object.

Evaluate `Point inspect`.

In Figure 17-5, the inspector shows the structure of class `Point`. You can see that the class stores its methods in a dictionary, indexing them by their selector. The selector `#*` points to the decompiled bytecode of `Point>>*`.

Let us consider the relationship between classes and methods. In Figure 17-6 we see that classes and metaclasses have the common superclass `Behavior`. This is where `new` is defined, amongst other key methods for classes. Every class has a method dictionary, which maps method selectors to compiled methods. Each compiled method knows the class in which it is installed. In Figure 17-5 we can even see that this is stored in an association in `literal6`.

We can exploit the relationships between classes and methods to pose queries about the system. For example, to discover which methods are newly introduced in a given class, *i.e.*, do not override superclass methods, we can navigate from the class to the method dictionary as follows:

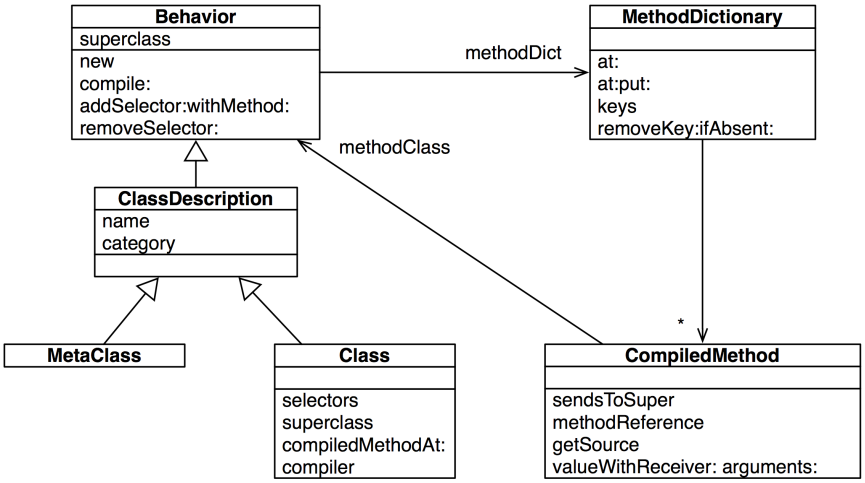

Figure 17-6 Classes, method dictionaries and compiled methods

```
[:aClass| aClass methodDict keys select: [:aMethod |
  (aClass superclass canUnderstand: aMethod) not ]] value:
    SmallInteger
>>> an IdentitySet(#threeDigitName #printStringBase:nDigits: ...)
```

A compiled method does not simply store the bytecode of a method. It is also an object that provides numerous useful methods for querying the system. One such method is `isAbstract` (which tells if the method sends `subclassResponsibility`). We can use it to identify all the abstract methods of an abstract class.

```
[:aClass| aClass methodDict keys select: [:aMethod |
  (aClass>>aMethod) isAbstract ]] value: Number
>>> #(#+ #round: #adaptToInteger:andSend: #asFloat #printOn:base: #/
    #adaptToFraction:andSend: #- #* #sqrt #nthRoot: #storeOn:base:)
```

Note that this code sends the >> message to a class to obtain the compiled method for a given selector.

To browse the super-sends within a given hierarchy, for example within the Collections hierarchy, we can pose a more sophisticated query:

```
class := Collection.
SystemNavigation default
  browseMessageList: (class withAllSubclasses gather: [:each |
    each methodDict associations
      select: [:assoc | assoc value sendsToSuper]
      thenCollect: [:assoc | RGMethodDefinition realClass: each
    selector: assoc key]])
  name: 'Supersends of ', class name, ' and its subclasses'
```

Note how we navigate from classes to method dictionaries to compiled methods to identify the methods we are interested in. A `RGMethodDefinition` is a lightweight proxy for a compiled method that is used by many tools. There is a convenience method `CompiledMethod>>methodReference` to return the method reference for a compiled method.

```
(Object>>#=) methodReference selector
>>> #=
```

17.4 Browsing environments

Although `SystemNavigation` offers some useful ways to programmatically query and browse system code, there are more ways. The Browser, which is integrated into Pharo, allows us to restrict the environment in which a search is to perform.

Suppose we are interested to discover which classes refer to the class `Point` but only in its own package.

Open a browser on the class `Point`.

Click on the top level package `Kernel` in the package pane and select `Scoped View` radio button. Browser now shows only the package `Kernel` and all classes within this package (and some classes which have extension methods from this package). Now, in this browser, select again the class `Point`, Action-click on the class name and select `Class refs`. This will show all methods that have references to the class `Point` but only those from the package `Kernel`. Compare this result with the search from a Browser without restricted scope.

This scope is what we call a *Browsing Environment* (class `RBBrowserEnvironment`). All other searches, like *senders of a method* or *implementors of a method* from within this browser are restricted to this environments too.

Browser environments can also be created programmatically. Here, for example, we create a new `RBBrowserEnvironment` for `Collection` and its subclasses, select the super-sending methods, and browse the resulting environment.

```
((RBBrowserEnvironment new forClasses: (Collection
    withAllSubclasses))
  selectMethods: [:method | method sendsToSuper])
  browse.
```

Note how this is considerably more compact than the earlier, equivalent example using `SystemNavigation`.

Finally, we can find just those methods that send a different super message programmatically as follows:

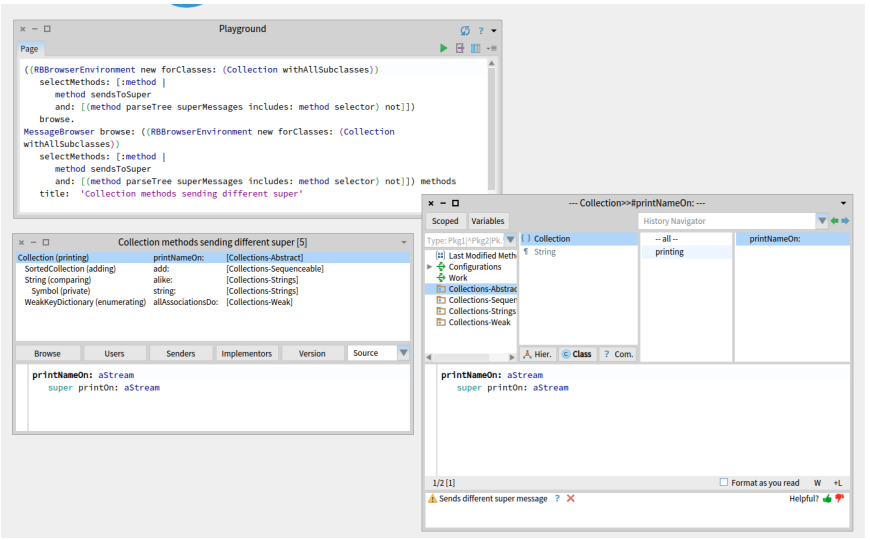

Figure 17-7 Finding methods

```
((RBBrowserEnvironment new forClasses: (Collection
    withAllSubclasses))
  selectMethods: [:method |
    method sendsToSuper
    and: [(method parseTree superMessages includes: method selector)
    not]])
  browse
```

Here we ask each compiled method for its (Refactoring Browser) parse tree, in order to find out whether the super messages differ from the method's selector. Have a look at the `querying` protocol of the class `RBProgramNode` to see some the things we can ask of parse trees.

Instead of browsing the environment in a System Browser, we can spawn a `MessageBrower` from the list of all methods in this environment.

```
MessageBrowser browse: ((RBBrowserEnvironment new forClasses:
    (Collection withAllSubclasses))
  selectMethods: [:method |
    method sendsToSuper
    and: [(method parseTree superMessages includes: method selector)
    not]]) methods
  title:  'Collection methods sending different super'
```

In Figure 17-7 we can see that 5 such methods have been found within the `Collection` hierarchy, including `Collection>>printNameOn:`, which sends `super printOn:`.

17.5 Accessing the run-time context

We have seen how Pharo's reflective capabilities let us query and explore objects, classes and methods. But what about the run-time environment?

Method contexts

In fact, the run-time context of an executing method is in the virtual machine — it is not in the image at all! On the other hand, the debugger obviously has access to this information, and we can happily explore the run-time context, just like any other object. How is this possible?

Actually, there is nothing magical about the debugger. The secret is the pseudo-variable `thisContext`, which we have encountered only in passing before. Whenever `thisContext` is referred to in a running method, the entire run-time context of that method is reified and made available to the image as a series of chained `Context` objects.

We can easily experiment with this mechanism ourselves.

Change the definition of `Integer>>factorial` by inserting the expression `thisContext inspect. self halt.` as shown below:

```
Integer>>factorial
  "Answer the factorial of the receiver."
  self = 0 ifTrue: [thisContext inspect. self halt. ^ 1].
  self > 0 ifTrue: [^ self * (self - 1) factorial].
  self error: 'Not valid for negative integers'
```

Now evaluate `3 factorial` in a workspace. You should obtain both a debugger window and an inspector, as shown in Figure 17-8.

Inspecting `thisContext` gives you full access to the current execution context, the stack, the local tempories and arguments, the senders chain and the receiver. Welcome to the poor man's debugger! If you now browse the class of the explored object (*i.e.*, by evaluating `self browse` in the bottom pane of the inspector) you will discover that it is an instance of the class `Context`, as is each `sender` in the chain.

`thisContext` is not intended to be used for day-to-day programming, but it is essential for implementing tools like debuggers, and for accessing information about the call stack. You can evaluate the following expression to discover which methods make use of `thisContext`:

```
SystemNavigation default browseMethodsWithSourceString:
    'thisContext' matchCase: true
```

As it turns out, one of the most common applications is to discover the sender of a message. Here is a typical application:

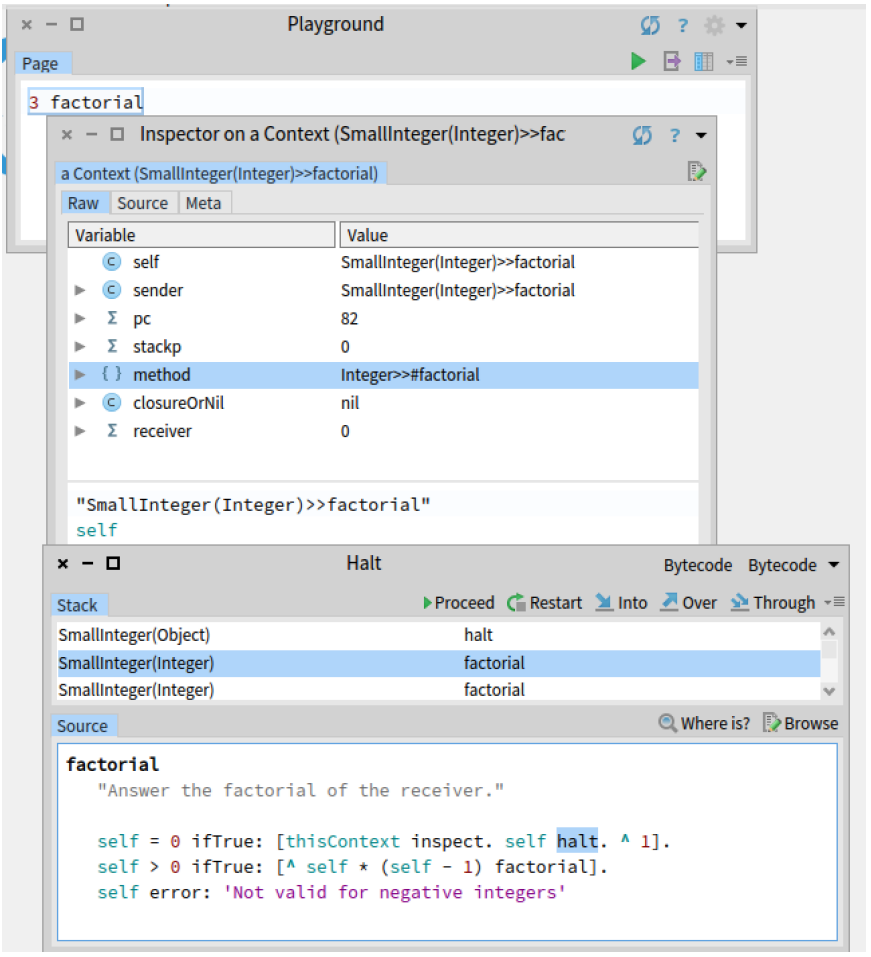

Figure 17-8 Inspecting `thisContext`.

```
subclassResponsibility
  "This message sets up a framework for the behavior of the class'
    subclasses.
  Announce that the subclass should have implemented this message."

  SubclassResponsibility signalFor: thisContext sender selector
```

By convention, methods that send `self subclassResponsibility` are considered to be abstract. But how does `Object>>subclassResponsibility` provide a useful error message indicating which abstract method has been invoked? Very simply, by asking `thisContext` for the sender.

Intelligent breakpoints

The Pharo way to set a breakpoint is to evaluate `self halt` at an interesting point in a method. This will cause `thisContext` to be reified, and a debugger window will open at the breakpoint. Unfortunately this poses problems for methods that are intensively used in the system.

Suppose, for instance, that we want to explore the execution of `Morph>>openInWorld`. Setting a breakpoint in this method is problematic.

Pay attention the following experiment will break everything! Take a *fresh* image and set the following breakpoint:

```
Morph >> openInWorld
     "Add this morph to the world."
     self halt.
     self openInWorld: self currentWorld
```

Notice how your image immediately freezes as soon as you try to open any new Morph (Menu/Window/...)! We do not even get a debugger window. The problem is clear once we understand that 1) `Morph>>openInWorld` is used by many parts of the system, so the breakpoint is triggered very soon after we interact with the user interface, but 2) *the debugger itself* sends `openInWorld` as soon as it opens a window, preventing the debugger from opening! What we need is a way to *conditionally halt* only if we are in a context of interest. This is exactly what `Object>>haltIf:` offers.

Suppose now that we only want to halt if `openInWorld` is sent from, say, the context of `MorphTest>>testOpenInWorld`.

Fire up a fresh image again, and set the following breakpoint:

```
Morph>>openInWorld
   "Add this morph to the world."
   self haltIf: #testOpenInWorld.
   self openInWorld: self currentWorld
```

This time the image does not freeze. Try running the `MorphTest`.

```
MorphTest run:#testOpenInWorld.
```

How does this work? Let's have a look at `Object>>haltIf:`. It first calls `if:` with the condition to the `Exception` class `Halt`. This method itself will check if the condition is a symbol, which is true in this case and finally calls

```
Object>>haltIf: condition
  <debuggerCompleteToSender>
  Halt if: condition.
```

```
Halt class >> haltIfCallChain: haltSenderContext contains: aSelector
  | cntxt |
  cntxt := haltSenderContext.
  [ cntxt isNil ] whileFalse: [
    cntxt selector = aSelector ifTrue: [ self signalIn:
    haltSenderContext ].
    cntxt := cntxt sender ]
```

Starting from `thisContext`, `haltIfCallChainContains:` goes up through the execution stack, checking if the name of the calling method is the same as the one passed as parameter. If this is the case, then it signals itself, the exception which, by default, summons the debugger.

It is also possible to supply a boolean or a boolean block as an argument to `haltIf:`, but these cases are straightforward and do not make use of `thisContext`.

17.6 Intercepting messages not understood

So far we have used Pharo's reflective features mainly to query and explore objects, classes, methods and the run-time stack. Now we will look at how to use our knowledge of its system structure to intercept messages and modify behaviour at run time.

When an object receives a message, it first looks in the method dictionary of its class for a corresponding method to respond to the message. If no such method exists, it will continue looking up the class hierarchy, until it reaches `Object`. If still no method is found for that message, the object will *send itself* the message `doesNotUnderstand:` with the message selector as its argument. The process then starts all over again, until `Object>>doesNotUnderstand:` is found, and the debugger is launched.

But what if `doesNotUnderstand:` is overridden by one of the subclasses of `Object` in the lookup path? As it turns out, this is a convenient way of realizing certain kinds of very dynamic behaviour. An object that does not understand a message can, by overriding `doesNotUnderstand:`, fall back to an alternative strategy for responding to that message.

Two very common applications of this technique are 1) to implement lightweight proxies for objects, and 2) to dynamically compile or load missing code.

Lightweight proxies

In the first case, we introduce a *minimal object* to act as a proxy for an existing object. Since the proxy will implement virtually no methods of its own, any message sent to it will be trapped by `doesNotUnderstand:`. By implementing this message, the proxy can then take special action before delegating the message to the real subject it is the proxy for.

Let us have a look at how this may be implemented.

We define a `LoggingProxy` as follows:

```
ProtoObject subclass: #LoggingProxy
  instanceVariableNames: 'subject invocationCount'
  classVariableNames: ''
  package: 'PBE-Reflection'
```

Note that we subclass `ProtoObject` rather than `Object` because we do not want our proxy to inherit around 450 methods (!) from `Object`.

```
Object methodDict size
>>> 440
```

Our proxy has two instance variables: the `subject` it is a proxy for, and a `count` of the number of messages it has intercepted. We initialize the two instance variables and we provide an accessor for the message count. Initially the `subject` variable points to the proxy object itself.

```
LoggingProxy >> initialize
  invocationCount := 0.
  subject := self.
```

```
LoggingProxy >> invocationCount
  ^ invocationCount
```

We simply intercept all messages not understood, print them to the `Transcript`, update the message count, and forward the message to the real subject.

```
LoggingProxy >> doesNotUnderstand: aMessage
  Transcript show: 'performing ', aMessage printString; cr.
  invocationCount := invocationCount + 1.
  ^ aMessage sendTo: subject
```

Here comes a bit of magic. We create a new `Point` object and a new `LoggingProxy` object, and then we tell the proxy to `become:` the point object:

```
point := 1@2.
LoggingProxy new become: point.
```

This has the effect of swapping all references in the image to the point to now refer to the proxy, and vice versa. Most importantly, the proxy's `subject` instance variable will now refer to the point!

```
point invocationCount
>>> 0
```

```
point + (3@4)
>>> 4@6
```

```
point invocationCount
>>> 1
```

This works nicely in most cases, but there are some shortcomings:

```
point class
>>> LoggingProxy
```

Actually the method `class` is implemented in `ProtoObject`, but even if it were implemented in `Object`, which `LoggingProxy` does not inherit from, it isn't actually send to the LoggingProxy or its subject. The message is directly answered by the virtual machine. `yourself` is also never truly sent.

Other messages that may be directly interpreted by the VM, depending on the receiver, include:

`+- < > <= >= = ~= * / \ == @ bitShift: // bitAnd: bitOr: at: at:put: size next nextPut: atEnd blockCopy: value value: do: new new: x y.`

Selectors that are never sent, because they are inlined by the compiler and transformed to comparison and jump bytecodes:

`ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue: and: or: whileFalse: whileTrue: whileFalse whileTrue to:do: to:by:do: caseOf: caseOf:otherwise: ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil:`

Attempts to send these messages to non-boolean normally results in an exception from the VM as it can not use the inlined dispatching for non-boolean receivers. You can intercept this and define the proper behavior by overriding `mustBeBoolean` in the receiver or by catching the `NonBooleanReceiver` exception.

Even if we can ignore such special message sends, there is another fundamental problem which cannot be overcome by this approach: `self`-sends cannot be intercepted:

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount
>>> 0
```

```
point rectangle: (3@4)
>>> 1@2 corner: 3@4
```

```
point invocationCount
>>> 1
```

Our proxy has been cheated out of two `self`-sends in the `rectangle:` method:

```
Point >> rectangle: aPoint
  "Answer a Rectangle that encompasses the receiver and aPoint. This
    is the most general infix way to create a rectangle."

  ^ Rectangle
    point: self
    point: aPoint
```

Although messages can be intercepted by proxies using this technique, one should be aware of the inherent limitations of using a proxy. In Section 17.7 we will see another, more general approach for intercepting messages.

Generating missing methods

The other most common application of intercepting not understood messages is to dynamically load or generate the missing methods. Consider a very large library of classes with many methods. Instead of loading the entire library, we could load a stub for each class in the library. The stubs know where to find the source code of all their methods. The stubs simply trap all messages not understood, and dynamically load the missing methods on demand. At some point, this behaviour can be deactivated, and the loaded code can be saved as the minimal necessary subset for the client application.

Let us look at a simple variant of this technique where we have a class that automatically adds accessors for its instance variables on demand:

```
Object subclass: #DynamicAccessors
  instanceVariableNames: 'x'
  classVariableNames: ''
  package: 'PBE-Reflection'
```

```
DynamicAcccessors >> doesNotUnderstand: aMessage
  | messageName |
  messageName := aMessage selector asString.
  (self class instVarNames includes: messageName)
    ifTrue: [
      self class compile: messageName, String cr, ' ^ ', messageName.
      ^ aMessage sendTo: self ].
  ^ super doesNotUnderstand: aMessage
```

Any message not understood is trapped here. If an instance variable with the same name as the message sent exists, then we ask our class to compile an accessor for that instance variables and we re-send the message.

Suppose the class `DynamicAccessors` has an (uninitialized) instance variable `x` but no pre-defined accessor. Then the following will generate the accessor dynamically and retrieve the value:

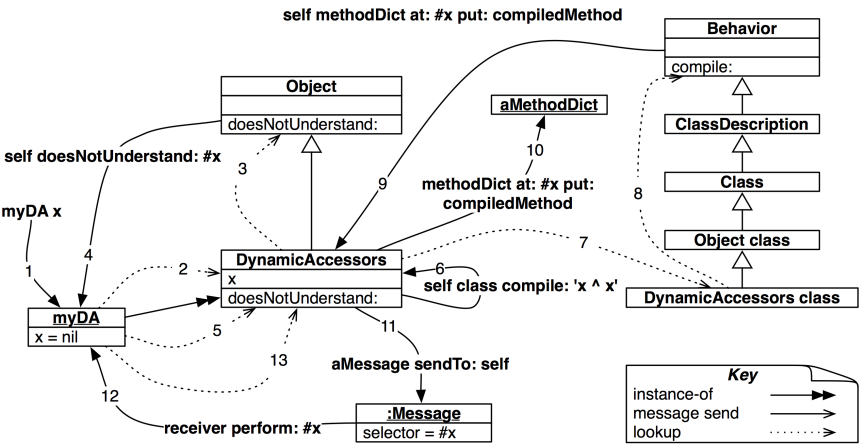

Figure 17-9 Dynamically creating accessors.

```
myDA := DynamicAccessors new.
myDA x
>>> nil
```

Let us step through what happens the first time the message `x` is sent to our object (see Figure 17-9).

(1) We send `x` to `myDA`, (2) the message is looked up in the class, and (3) not found in the class hierarchy. (4) This causes `self doesNotUnderstand: #x` to be sent back to the object, (5) triggering a new lookup. This time `doesNotUnderstand:` is found immediately in `DynamicAccessors`, (6) which asks its class to compile the string `'x ^ x'`. The `compile` method is looked up (7), and (8) finally found in `Behavior`, which (9-10) adds the new compiled method to the method dictionary of `DynamicAccessors`. Finally, (11-13) the message is resent, and this time it is found.

The same technique can be used to generate setters for instance variables, or other kinds of boilerplate code, such as visiting methods for a Visitor.

Note the use of `Object>>perform:` in step (12) which can be used to send messages that are composed at run-time:

```
5 perform: #factorial
>>> 120
```

```
6 perform: ('fac', 'torial') asSymbol
>>> 720
```

```
4 perform: #max: withArguments: (Array with: 6)
>>> 6
```

17.7 Objects as method wrappers

We have already seen that compiled methods are ordinary objects in Pharo, and they support a number of methods that allow the programmer to query the runtime system. What is perhaps a bit more surprising, is that *any object* can play the role of a compiled method. All it has to do is respond to the method `run:with:in:` and a few other important messages.

Define an empty class `Demo`. Evaluate `Demo new answer42` and notice how the usual *Message Not Understood* error is raised.

```
Object subclass: #Demo
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-Reflection'
```

```
Demo new answer42
```

Now we will install a plain object in the method dictionary of our `Demo` class.

Evaluate `Demo methodDict at: #answer42 put: ObjectsAsMethodsExample new.`

Now try again to print the result of `Demo new answer42`. This time we get the answer `42`.

```
Demo methodDict at: #answer42 put: ObjectsAsMethodsExample new.
Demo new answer42
>>> 42
```

If we take look at the class `ObjectsAsMethodsExample` we will find the following methods:

```
add: a with: b
  ^a + b

answer42
  ^42

run: oldSelector with: arguments in: aReceiver
  ^self perform: oldSelector withArguments: arguments
```

When our `Demo` instance receives the message `answer42`, method lookup proceeds as usual, however the virtual machine will detect that in place of a compiled method, an ordinary Pharo object is trying to play this role. The VM will then send this object a new message `run:with:in:` with the original method selector, arguments and receiver as arguments. Since `ObjectsAsMethodsExample` implements this method, it intercepts the message and delegates it to itself.

We can now remove the fake method as follows:

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

If we take a closer look at `ObjectsAsMethodsExample`, we will see that its superclass also implements some methods like `flushcache`, `methodClass:` and `selector:`, but they are all empty. These messages may be sent to a compiled method, so they need to be implemented by an object pretending to be a compiled method. (`flushcache` is the most important method to be implemented; others may be required by some tools and depending on whether the method is installed using `Behavior>>addSelector:withMethod:` or directly using `MethodDictionary>>at:put:`.)

NEXT CHAPTER NEEDS A REVIEW!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Using method wrappers to perform test coverage

Method wrappers are a well-known technique for intercepting messages. In the original implementation(http://www.squeaksource.com/MethodWrappers.html), a method wrapper is an instance of a subclass of `CompiledMethod`. When installed, a method wrapper can perform special actions before or after invoking the original method. When uninstalled, the original method is returned to its rightful position in the method dictionary.

In Pharo, method wrappers can be implemented more easily by implementing `run:with:in:` instead of by subclassing `CompiledMethod`. In fact, there exists a lightweight implementation of objects as method wrappers (http://www.squeaksource.com/ObjectsAsMethodsWrap.html), but it is not part of standard Pharo at the time of this writing.

Nevertheless, the Pharo Test Runner uses precisely this technique to evaluate test coverage. Let's have a quick look at how it works.

The entry point for test coverage is the method `TestRunner>>runCoverage`:

```
TestRunner >> runCoverage
  | packages methods |
  ... "identify methods to check for coverage"
  self collectCoverageFor: methods
```

The method `TestRunner>>collectCoverageFor:` clearly illustrates the coverage checking algorithm:

```
TestRunner >> collectCoverageFor: methods
  | suite link notExecuted |
  suite := self
    resetResult;
    suiteForAllSelected.

  link := MetaLink new
    selector: #tagExecuted;
    metaObject: #node.

  [ methods do: [ :meth | meth ast link: link].
```

```
  [ self runSuite: suite ] ensure: [ link uninstall ] ]
    valueUnpreemptively.

  notExecuted := methods reject: [:each | each ast hasBeenExecuted].
  notExecuted isEmpty
    ifTrue:  [ UIManager default inform: 'Congratulations. Your
    tests cover all code under analysis.' ]
    ifFalse: ...
```

CODE NOT UPDATED FROM HERE TO NEXT EXCALAMATION!!!!!!!!!!!!!!!!!! A wrapper is created for each method to be checked, and each wrapper is installed. The tests are run, and all wrappers are uninstalled. Finally the user obtains feedback concerning the methods that have not been covered.

How does the wrapper itself work? The `TestCoverage` wrapper has three instance variables, `hasRun`, `reference` and `method`. They are initialized as follows:

```
TestCoverage class >> on: aMethodReference
  ^ self new initializeOn: aMethodReference

TestCoverage >> initializeOn: aMethodReference
  hasRun := false.
  reference := aMethodReference.
  method := reference compiledMethod
```

The install and uninstall methods simply update the method dictionary in the obvious way:

```
TestCoverage >> install
  reference actualClass methodDict
    at: reference selector
    put: self

TestCoverage >> uninstall
  reference actualClass methodDict
    at: reference selector
    put: method
```

The `run:with:in:` method simply updates the `hasRun` variable, uninstalls the wrapper (since coverage has been verified), and resends the message to the original method.

```
run: aSelector with: anArray in: aReceiver
  self mark; uninstall.
  ^ aReceiver withArgs: anArray executeMethod: method

mark
  hasRun := true
```

Take a look at `ProtoObject>>withArgs:executeMethod:` to see how a method displaced from its method dictionary can be invoked.

That's all there is to it!

Method wrappers can be used to perform any kind of suitable behaviour before or after the normal operation of a method. Typical applications are instrumentation (collecting statistics about the calling patterns of methods), checking optional pre- and post-conditions, and memoization (optionally cacheing computed values of methods).

CORRECTION IN THE SUMMARY AS WELL!!!!!!!!!!!!!!!!!!!!!!!!!!!!

17.8 Pragmas

A *pragma* is an annotation that specifies data about a program, but is not involved in the execution of the program. Pragmas have no direct effect on the operation of the method they annotate. Pragmas have a number of uses, among them:

Information for the compiler: pragmas can be used by the compiler to make a method call a primitive function. This function has to be defined by the virtual machine or by an external plug-in.

Runtime processing: Some pragmas are available to be examined at runtime.

Pragmas can be applied to a program's method declarations only. A method may declare one or more pragmas, and the pragmas have to be declared prior any Smalltalk statement. Each pragma is in effect a static message send with literal arguments.

We briefly saw pragmas when we introduced primitives earlier in this chapter. A primitive is nothing more than a pragma declaration. Consider `<primitive: 173 error:ec>` as contained in `instVarAt:`. The pragma's selector is `primitive:error:` and its arguments is an immediate literal value, `173`. The variable `ec` is an error code, filled by the VM in case the execution of the implementation on the VM side failed.

The compiler is probably the bigger user of pragmas. SUnit is another tool that makes use of annotations. SUnit is able to estimate the coverage of an application from a test unit. One may want to exclude some methods from the coverage. This is the case of the `documentation` method in `SplitJointTest class`:

```
SplitJointTest class >> documentation
  <ignoreForCoverage>
  "self showDocumentation"

  ^ 'This package provides function.... '
```

By simply annotating a method with the pragma `<ignoreForCoverage>` one can control the scope of the coverage.

As instances of the class `Pragma`, pragmas are first class objects. A compiled method answers to the message `pragmas`. This method returns an array of pragmas.

```
(SplitJoinTest class >> #documentation) pragmas.
>>> an Array(<ignoreForCoverage>)
```

```
(SmallFloat64>>#+) pragmas
>>> an Array(<primitive: 541>)
```

Methods defining a particular query may be retrieved from a class. The class side of `SplitJoinTest` contains some methods annotated with `<ignoreForCoverage>`:

```
Pragma allNamed: #ignoreForCoverage in: SplitJoinTest class
>>> an Array(<ignoreForCoverage> <ignoreForCoverage>)
```

A variant of `allNamed:in:` may be found on the class side of `Pragma`.

A pragma knows in which method it is defined (using `method`), the name of the method (`selector`), the class that contains the method (`methodClass`), its number of arguments (`numArgs`), about the literals the pragma has for arguments (`hasLiteral:` and `hasLiteralSuchThat:`).

17.9 Chapter summary

Reflection refers to the ability to query, examine and even modify the metaobjects of the runtime system as ordinary objects.

- The Inspector uses `instVarAt:` and related methods to view *private* instance variables of objects.
- Send `Behavior>>allInstances` to query instances of a class.
- The messages `class`, `isKindOf:`, `respondsTo:` etc. are useful for gathering metrics or building development tools, but they should be avoided in regular applications: they violate the encapsulation of objects and make your code harder to understand and maintain.
- `SystemNavigation` is a utility class holding many useful queries for navigation and browsing the class hierarchy. For example, use `SystemNavigation default browseMethodsWithSourceString: 'pharo'`

`matchCase:true.` to find and browse all methods with a given source string. (Slow, but thorough!)

- Every Pharo class points to an instance of `MethodDictionary` which maps selectors to instances of `CompiledMethod`. A compiled method knows its class, closing the loop.
- `RGMethodDefinition` is a leightweight proxy for a compiled method, providing additional convenience methods, and used by many Pharo tools.
- `RBBrowserEnvironment`, part of the Refactoring Browser infrastructure, offers a more refined interface than `SystemNavigation` for querying the system, since the result of a query can be used as a the scope of a new query. Both GUI and programmatic interfaces are available.
- `thisContext` is a pseudo-variable that reifies the runtime stack of the virtual machine. It is mainly used by the debugger to dynamically construct an interactive view of the stack. It is also especially useful for dynamically determining the sender of a message.
- Intelligent breakpoints can be set using `haltIf:`, taking a method selector as its argument. `haltIf:` halts only if the named method occurs as a sender in the run-time stack.
- A common way to intercept messages sent to a given target is to use a *minimal object* as a proxy for that target. The proxy implements as few methods as possible, and traps all message sends by implementing `doesNotunderstand:`. It can then perform some additional action and then forward the message to the original target.
- Send `become:` to swap the references of two objects, such as a proxy and its target.
- Beware, some messages, like `class` and `yourself` are never really sent, but are interpreted by the VM. Others, like `+`, `-` and `ifTrue:` may be directly interpreted or inlined by the VM depending on the receiver.
- Another typical use for overriding `doesNotUnderstand:` is to lazily load or compile missing methods.
- `doesNotUnderstand:` cannot trap `self`-sends.
- A more rigorous way to intercept messages is to use an object as a method wrapper. Such an object is installed in a method dictionary in place of a compiled method. It should implement `run:with:in:` which is sent by the VM when it detects an ordinary object instead of a compiled method in the method dictionary. This technique is used by the SUnit Test Runner to collect coverage data.