The Cuis-Smalltalk book

Introduction to Programming with Smalltalk

Hilaire Fernandes with Ken Dickey & Juan Vuletich

Short Contents

Table of Contents

Preface

> A language that doesn't affect the way you think about programming, is not worth knowing.
> —*Alan Perlis*

Cuis-Smalltalk —in short, Cuis— is a portable environment for doing, building and sharing software. Like any other tool, Cuis doesn't require having a particular mindset or following a particular process. However, Cuis is built with a specific view on what software is, and what it means to build software. As a consequence, Cuis is especially effective if these ideas resonate with you and, at least occasionally, you let them guide your thoughts and actions.

In this book, we will accompany you as you explore, discover and learn about Cuis and Smalltalk.

We don't assume knowledge of computer programming, but if you happen to have some experience with an Object-Oriented or Functional language you will recognize many concepts. If you don't have any programming experience or you have coded in an imperative, closer to the metal language, such as C or Assembly, many of the ideas may be new.

In any case, especially during the first chapters, read this book and follow the examples with an open mind. A new point of view on what software development means will be an enriching experience. For us, programming is a thoughtful process.

We understand software development as the activity of learning and documenting knowledge for ourselves and others to use, and also to be run on a computer. The fact that a computer can execute the software and produce useful solutions to some problem is a consequence of the knowledge being sound and relevant. Just making it work is not the important part!

These ideas strongly shape Cuis and the experience of using it, and we'll develop them further along in this book. Amongst their most obvious consequences are a passion for reducing unneeded complexity, while providing a complete, live software development experience.

This book is an introduction and invitation to explore Cuis. We hope you will join us in this journey to use it as a medium to express ideas and thoughts, to build cool stuff, and to make Cuis even better.
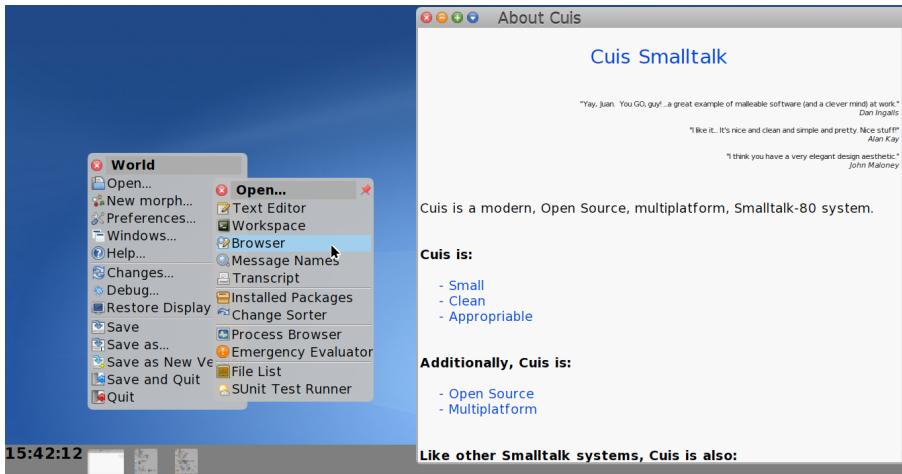
Figure 1: Cuis

To make your journey with this book more enjoyable, the *Spacewar!*[1] project is its recurring theme. It is distilled into the book in code examples, exercises and dedicated chapters. At the end of the book, you will have written a replica of this historic video game. The final Cuis-Smalltalk package of this game can be downloaded from the book repository[2].

How to use the book

The reader can study Cuis-Smalltalk from two versions of the book:

- **Online.** With a web browser at the `https://cuis-smalltalk.github.io/TheCuisBook` address. The book is structured in chapters and sections, displayed one at a time. It is a very flexible way to study the book: you can open several chapters and appendices on different tabs in your web browser. It requires you to be connected to the Internet, though.

- **Offline.** In a single PDF file you download and read offline. It can be printed as a nice paper book. The most recent build can be found at `https://github.com/Cuis-Smalltalk/TheCuisBook`.

---

[1] `https://en.wikipedia.org/wiki/Spacewar!`
[2] `https://github.com/Cuis-Smalltalk/TheCuisBook/blob/master/Spacewar!.pck.st`

The chapters come with many examples. Some can be copied and pasted into Cuis-Smalltalk. We encourage you to do this, and in the process modify them to explore on your own.

The code examples in the online version can be directly copied and pasted into Cuis-Smalltalk. This is why the assignment character "←" you see in the developer Cuis-Smalltalk window is printed as ":=" in the online version of the book. The same applies with the return character "↑" printed as "^" in the online version.

In the offline PDF version, the code examples are printed with the same assignment and return characters as seen in the Cuis-Smalltalk windows. Copying and pasting also work as expected.

Other examples are code extracts which are not self sufficient to be executed as is. Their purpose is to expose specific facets of the Smalltalk language.

A typical example without a caption looks like:

```
100 factorial
```

The same example with a caption appears with a number and title, and can be used as a reference elsewhere in the book:

```
100 factorial
⇒  93326215443944152681699238856266700490715968264438162146859
29638952175999932299156089414639761565182862536979208272237582
51185210916864000000000000000000000000000
```

Example 1: I am an example with a caption and a result

There are also a lot of exercises. Most are very easy; they intend to give you an opportunity to apply what you have just learned.

Exercises are easily identified by the Cuis-Smalltalk mascot: *Cuis*, a South American rodent!

Search the Internet: How many versions of Smalltalk are there?

Exercise 1: I am an example of an exercise

The solutions to the exercises are presented in Appendix D [Solutions to the Exercises], page 184.

Happy reading!

1 Beginnings

> The computer is simply an instrument whose music is ideas.
> —*Alan Kay*

Before getting into the details of how to use the Cuis-Smalltalk language and tools to build software, we need to understand the point of view, assumptions and intentions that shape how Cuis-Smalltalk is meant to be used. We can call this the Smalltalk philosophy of programming.

1.1 Historical Context

One major idea in software is that programming is merely giving a computer a set of instructions to solve some problem. In this point of view, the only value of software is to achieve a result, and therefore, a piece of software is only as interesting as having that problem solved. Furthermore, as programming is not interesting by itself, it is left to a professional guild with a specialized technical knowledge of how to write software, and the rest of the world just ignores it.

Our focus is on one thread in the history of powerful ideas which give a different lens through which to explore software development. The historical development of these ideas differs from the *just solve a problem* view. We think this is worth revisiting.

The first clear vision of an automated information processor to augment our collective memory, to find and share knowledge – indeed to transform the data explosion into an information explosion, and then into a knowledge and understanding explosion – was called Memex and described in the Vannevar Bush essay "As We May Think" in 1945[1]. Bush's description of the possibility of developing a Memex processing system to help individuals access, evolve and capture knowledge for our collective benefit inspired many later thinkers and inventors in the development of personal computers, networks, hypertext, search engines, and knowledge repositories such as Wikipedia[2].

As computational machinery evolved from large, single program at a time mainframes, into timesharing mainframes, and into minicomputers, another

---

[1] `https://en.wikipedia.org/wiki/As_We_May_Think`

[2] Notable early milestones in this line of development were Ivan Sutherland's Sketchpad (1963), the RAND tablet (1964), and Doug Engelbart's NLS ("oN-Line System") (1968). Later developments include Ted Nelson's Xanadu, the Apple Macintosh, the World Wide Web, smartphones and tablet computers.

area of insight was developing with models of how humans learn[3]. People began to talk of *human-computer symbiosis*. Alan Kay conceived the idea that computer software and computer programming could become a new medium for expressing thoughts and knowledge. The ability to express ideas in this new medium would be the new literacy. Every person should learn to read and write in this new digital medium, and would have available their own dynamic book, a *Dynabook*[45] to accomplish this.

Very readable accounts of historic developments are Alan Kay's papers "The Early History of Smalltalk"[6] and "What is a Dynabook?"[7] in which he notes *The best way to predict the future is to invent it*. Realizing the Dynabook vision required significant advances and mutually supporting development of hardware and software. The original development took place at the Xerox PARC research center in the 1970's. The first *interim Dynabook* hardware was the Xerox Alto, generally considered the first personal computer. The software system was Smalltalk. The design of both was guided by the objectives of making *Personal Dynamic Media* and the Dynabook real. The final version of Smalltalk built at Xerox was Smalltalk-80.

Given the relatively weak capabilities of computer hardware at that time, implementing this vision presented real challenges, and much creativity was called for. Today, smartphones, tablets and laptops do have the hardware capabilities a Dynabook requires. However, the same advance hasn't happened for the medium of personal software.

In 1981 Dan Ingalls, one of the early Smalltalk inventors, wrote in his article "Design Principles Behind Smalltalk"[8] a number of principles that still guide us today. Among these:

> **Personal Mastery.** If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.

> **Reactive Principle.** Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.

With the commercialization of software, the trend has been to give people "shrink wrapped" applications which are sealed off and written by professional software developers. One may customize an application by changing "user settings", but there is no way to see into or change deep capabilities.

---

[3] Early notables here are J. Piaget, J. Brunner, O. K. Moore, and S. Papert.

[4] "A personal computer for children of all ages"(1972) `http://www.vpri.org/pdf/hc_pers_comp_for_children.pdf`

[5] "Personal Dynamic Media"(1977) `http://www.vpri.org/pdf/m1977001_dynamedia.pdf`

[6] `http://worrydream.com/EarlyHistoryOfSmalltalk`

[7] `http://www.vpri.org/pdf/hc_what_Is_a_dynabook.pdf`

[8] `http://www.cs.virginia.edu/~evans/cs655/readings/smalltalk.html`

In keeping with the ideal of personal media literacy, we believe that everyone should have full access to the software that runs our systems. Understanding and exploring computer systems requires writing software in ways that can be read and shared.

The thinking change in problem solving can shift from *What can I do here with what I am presented?* to asking *What tools do I need to be successful here?* and then building them to be ever more successful.

This way of thinking about software systems is important to us.

For this reason, Cuis is a *kernel* Smalltalk system which is still rather close to Smalltalk-80. The Cuis-Smalltalk goal is to be a small, coherent Smalltalk development environment that, with study, is comprehensible to a single person.

As we experiment with and evolve Cuis, this goal is carried out by removing everything possible from the base system which is not essential, and by having a composition strategy which allows one to write or add any features as needed. As one gains understanding of the software kernel or core, one only has to read and learn from each additional feature at a time to understand the whole.

Modern, open software environments are highly complex. Cuis is an attempt to remain oriented and able to discover patterns without being lost in a large wealth of possibilities which one does not completely grasp.

It is said that one understands the world by building it. Developing fluency and depth in a new medium takes time and practice.

We invite you to become fluent and look forward to sharing with us what you have built.

First, however, we must start with some of the mechanics.

As the saying goes "A journey of a thousand miles begins with a single step"[9].

Let's step forward together.

1.2 Installing and configuring Cuis-Smalltalk

Cuis-Smalltalk is an environment and a programming language executed on an idealized virtual computer. It is based on two major components: the Smalltalk *virtual machine* conceptualizing this virtual computer and an *image* representing the state of this computer.

The virtual machine is an executable program running on a dedicated host (GNU/Linux, Mac OS X, Windows, etc.). It is called the *Open Smalltalk Virtual Machine*, or Squeak VM in short. There are different flavors of

---

[9] `https://en.wikipedia.org/wiki/A_journey_of_a_thousand_miles_begins_with_a_single_step`

VM, for various combinations of Operating System and CPU architecture. Therefore, one VM compiled for Windows on Intel architecture will not work in Linux on ARM architecture. You need the specific VM compiled for the combination of Operating System and CPU architecture your computer is based on.

The *image* is a regular file feeding the VM with all the objects defining the state of the virtual computer. These objects are classes, methods, instances of those classes as numbers, strings, windows, debuggers – whatever existed when the state of the virtual computer was saved. An *image file* saved on a given Operating System and CPU architecture **will run identically on another system** requiring only a compatible VM to be used.

The VM allows an image to be restarted with windows in the same locations between different operating systems and different CPU architectures without recompilation. This is what *portability* means to us.

What makes Cuis-Smalltalk special is the living entities in the image: its class population and arrangement, how the classes inherit from each other. The class count is typically less than 700.

To get you started easily, we encourage you to install Cuis-University[10]. Here you will find bundles for GNU/Linux, Mac OS X and Windows on Intel architecture. These bundles come with the dedicated VM and a recent image of Cuis-Smalltalk; as well packages ready to be installed to make your life easier when you experience the examples and exercises of the book. By the time you read this book, Cuis-Smalltalk will likely have evolved to a newer version. What you learn here should, however, be easily transferable.

To get Cuis-Smalltalk running on your computer, extract the bundle and execute the run script on Windows/Linux – `run.bat` or `run.sh` – on OS X launch the Squeak application. Once you get Cuis-Smalltalk running, read the information displayed on the windows. When you are done, you can close these windows and adjust Cuis-Smalltalk to your preferences.

The Cuis University distribution should work for most common platforms, but there are always more platform variants than we can test for. **If you have a problem**, here are two sources of information. If you do *not* have a problem, you can ignore these for now.

- Current installation instructions at the GitHub Cuis Repository: `https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev#setting-up-cuis-in-your-machine`

- Ask us on the Cuis email group `https://lists.cuis.st/mailman/listinfo/cuis-dev`

---

[10] `https://sites.google.com/view/cuis-university/descargas`

1.2.1 Editing your preferences

Once you read the information on the default windows, the next thing you want to do is to adjust visual properties to fit your preferences and needs. To do so, access the World menu ...Background click → `Preferences...` then select the pin on the top right of the menu to make it permanent. Here you have the most important options: the choice for the **font size**, the **themes** whenever your prefer light or dark colouring. There are other preferences you can explore by yourself. Once you are done, do ...World menu → `Save...` to make your preferences permanent. In this book, we keep the default Cuis-Smalltalk theme, we suggest you do the same so your environment reflects the book screenshots.
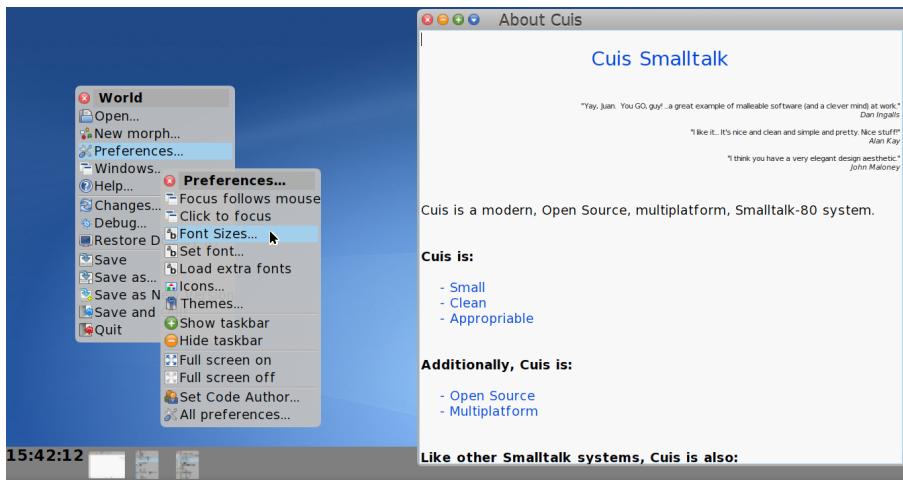
Figure 1.1: Set Preferences

1.2.2 Fun with window placement

The first tool to discover is the *Workspace* tool. It is a kind of text editor to key in Smalltalk code you can execute immediately. Do ...World menu → `Open...` → `Workspace`...

Now we will ask Cuis-Smalltalk to change the window's placement: click the blue icon (top left) to access the window's options menu and experiment with the white area to place the Workspace window at the half left of the Cuis-Smalltalk environment.
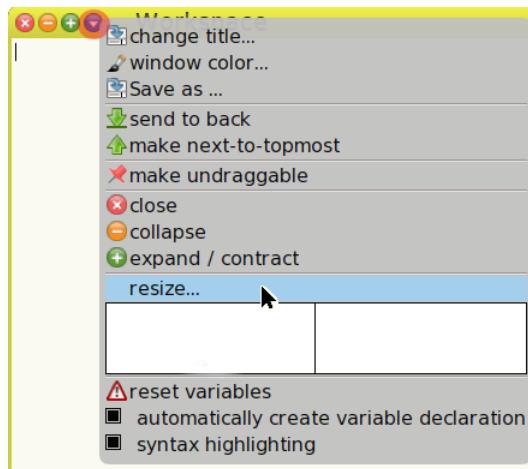
Figure 1.2: Window options

The `resize...` option offers even more freedom to place the window. Try the following exercise:

> Use the `resize...` option to place the Workspace centered in the Cuis-Smalltalk environment.

Exercise 1.1: Middle placement

1.3 Writing your first scripts

In this section you will learn how to write simple scripts in the Workspace to get a taste and feel for Smalltalk code. The examples are associated with small exercises to experiment with and accompanied with solutions in the appendix. We intentionally kept the details of the syntax out of this section.

In a Workspace, the usual *Hello World!* program can be written in Smalltalk:

```
Transcript show: 'Hello World!'
```

Example 1.1: The traditional 'Hello World!' program

To execute this code, select it with the mouse and over it do ...right mouse click → `Do it (d)`... You may now see nothing happen! Indeed to see the output, you need a *Transcript* window to be visible. The `Transcript` is a place where a programmer can send information to the user as we are doing here. Do ...World menu → `Open...` → `Transcript`... and execute the code again.

Figure 1.3: Transcript window with output

The workspace code has three parts:

- the string literal `'Hello World!'`
- the message `#show:` with its argument `'Hello World!'`
- the class `Transcript` receiving the message `#show:` with its argument

The action of printing takes place in the class `Transcript`. The code execution is also invoked with keyboard shortcuts `Ctrl-a` (*select* **A**ll) then `Ctrl-d` (**D**o *it*).

```
Transcript show: 'Hello World!'.
Transcript newLine.
Transcript show: 'I am Cuising'
```

Example 1.2: Multiple lines

In this three line script, observe how the lines are separated by a dot ".". This dot (or period) is a line separator so it is not needed in the third line nor in a one line script. The message `#newLine` has no argument.

A *String* is the way text is represented in a programming language, it is a collection of characters. We already met string with our first script, it is enclosed in single quotes: `'hello world!'`. We capitalize it with the `#capitalized` message:

```
Transcript show: 'hello world!' capitalized
⇒ 'Hello world!'
```

To convert all the characters to capitals use the `#asUppercase` message:

```
Transcript show: 'hello world!' asUppercase
```

```
⇒ 'HELLO WORLD!'
```

Two strings are concatenated with the **#,** message:

```
Transcript show: 'Hello ', 'my beloved ', 'friend'
⇒ 'Hello my beloved friend'
```

Example 1.3: Concatenate strings

Add a message to modify Example 1.3 to output 'Hello MY BELOVED friends'.

Exercise 1.2: Concatenate and uppercase

1.3.1 Fun with numbers

In your Workspace, to compute a factorial execute the example below with *Ctrl-a* then *Ctrl-p* (**P**rint it):

```
100 factorial
⇒  93326215443944152681699238856266700490715968264381621468592
9638952175999932299156089414639761565182862536979208272237582
5118521091686400000000000000000000000000
```

Cuis-Smalltalk handles very large integer numbers without requiring a special type declaration or method. To convince yourself try the example below:

```
10000 factorial / 9999 factorial
⇒ 10000
```

If you execute and print with *Ctrl-p* the code: `10000 factorial`, you will see that it takes far more time to print one factorial result than to compute two factorials and divide them. The result is an integer as expected, not a scaled decimal number as many computer languages will return.

As we are discussing division, you may not get the result you expect:

```
15 / 4
⇒ 15/4
```

It looks like Cuis-Smalltalk is lazy because it does not answer the decimal number `3.75` as we were expecting. In fact Cuis-Smalltalk wants to be as accurate as possible, and its answer is a rational number! After all, a fraction is just a division we are too lazy – because it is troublesome – to compute, Cuis-Smalltalk does just that!

Try out this to understand what is happening underneath:

```
(15 / 4) + (1 / 4)
⇒ 4
```

Is it not wonderful? Cuis-Smalltalk computes with rational numbers. We started with division and addition operations on integer, and we got an accurate result thanks to intermediate computation on rational numbers.

> In the example, observe how the parentheses are used although in arithmetic calculation the division is performed first. With Cuis-Smalltalk you need to specify the order of operations with parentheses. We will explain why later.

> *Write the code to compute the sum of the first four integer inverses. 4 inverted is 1/4*

Exercise 1.3: Inverse sum

Integers can be printed in different forms with the appropriate message:

```
2020 printStringRoman ⇒ 'MMXX'
2020 printStringWords ⇒ 'two thousand, twenty'
"Number as the Maya did"
2020 printStringBase: 20 ⇒ '510'
```

Print 2020 as words, capitalized.

Exercise 1.4: Capitalize number as words

The integer and float numbers we have played with are *Numeric Literals*. Literals are building blocks of the language and considered as constant expressions. They literally are as they appear.

There are several syntax variants which denote a number:

Numeric literal	What it represents
1	integer (decimal notation)
2r101	integer (binary radix)
16rFF	integer (hexadecimal radix)
1.5	floating point number
2.4e7	floating point (exponential notation)

Depending on the value we need to use, we can mix these literal representations:

```
16rA + 1 + 5e-1 + 6e-2
⇒ 289/25
```

1.4 Spacewar!

The Spacewar! game was initially developed in 1962 by Steve Russell on a DEC PDP-1 minicomputer. It is said to be the first known video game installed on several computers and it was very popular in the programming community in the 1960s. It was ported and rewritten several times to different hardware architectures and complemented with additional features. *Computer Space*, the first arcade video game cabinet was a clone of Spacewar!

Figure 1.4: Spacewar! game on a DEC PDP-1 minicomputer

Wikipedia describes very precisely this space combat simulation game:

The gameplay of Spacewar! involves two monochrome spaceships called "the needle" and "the wedge", each controlled by a player, attempting to shoot one another while maneuvering on a two-dimensional plane in the gravity well of a star, set against the backdrop of a starfield. The ships fire torpedoes, which are not affected by the gravitational pull of the star. The ships have a limited number of torpedoes and supply of fuel, which is used when the player fires the ship's thrusters. Torpedoes are fired one at a time by flipping a toggle switch on the computer or pressing a button on the control pad, and there is a cooldown period between launches. The ships remain in motion even when the player is not accelerating, and rotating the ships does not change the direction of their motion, though the ships can rotate at a constant rate without inertia.

Each player controls one of the ships and must attempt to shoot down the other ship while avoiding a collision with the star or the opposing ship. Flying near the star can provide a gravity assist to the player at the risk of misjudging the trajectory and falling into the star. If a ship moves past one edge of the screen, it reappears on the other side in a wraparound effect.
       —*Wikipedia, Spacewar!*

Therefore, the protagonists of the game are:

1. a **central star** generating a gravity field,

2. a **star field** background,

3. two **space ships** called *the needle* and *the wedge* controlled by two play-

ers.

4. **torpedoes** fired by the space ships.

Figure 1.5: Spacewar! game play

2 The Message Way of Life

> The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
> —*Alan Kay*

A Smalltalk system is a collection of entities communicating with each other through messages. That's all, there is nothing more.

2.1 Communicating entities

When a given entity receives a message from another entity, it triggers a specific behavior. The receiving entity of the message is called the *receiver* and the sending entity, the *sender*. In Cuis-Smalltalk terminology, an entity is called an *instance of a class*, a *class instance*, or simply an *instance*. A *class* is a kind of model for an entity.

The behavior is defined internally in the receiver and it can be triggered from any instance. **Behaviors are invoked only by messages sent between entities**. An entity may send a message to itself. A behavior is defined in a class and is called a *method*.

This results in a huge cloud of entities communicating with each other through message sending. New entities are instantiated when needed then automatically garbage collected when no longer required. On a fresh Cuis-Smalltalk environment, the count of class instances is more than 150000.

```
ProtoObject allSubclasses sum: [ :class | class allInstances size]
⇒ 152058
```

Example 2.1: Calculating the number of entities

The count of classes, the models for the entities – instances of the class `Class` – is less than 700.

```
Smalltalk allClasses size
⇒ 671
```

Example 2.2: Calculating the number of classes

> Because you are not using the base image but one used to teach classes, you will likely see a much larger number.

To be honest, in our previous chapter we skipped this important detail on Smalltalk design. We wrote about message sending without explaining much, we wanted you to discover this design informally. The scripts you read and wrote were all about entities communicating with each other through messages.

2.2 Message send definitions

There are three kinds of messages in Cuis-Smalltalk:

- **Unary messages** take no argument.
  In `1 factorial` the message `#factorial` is sent to the object `1`.
- **Binary messages** take exactly one argument.
  In `1 + 2` the message `#+` is sent to the object `1` with the argument `2`.
- **Keyword messages** take an arbitrary number of arguments.
  In `2 raisedTo: 6 modulo: 10` the message consisting of the message selector `#raisedTo:modulo:` and the arguments `6` and `10` is sent to the object `2`.

Unary message selectors consist of alphanumeric characters, and start with a lower case letter.

Binary message selectors consist of one or more characters from the following set:

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

Keyword message selectors consist of a series of alphanumeric keywords, where each keyword starts with a lower-case letter and ends with a colon.

Unary messages have the highest precedence, then binary messages, and finally keyword messages, so:

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

First we send `factorial` to `3`, then we send `+ 6` to `1`, and finally we send `raisedTo: 7` to `2`.

Precedence aside, evaluation is strictly from left to right, so

```
1 + 2 * 3
```

```
⇒ 9
```

is not 7. Parentheses must be used to alter the order of evaluation:

```
1 + (2 * 3)
⇒ 7
```

Parentheses can also be used to make potentially confusing code more clear. For example, the strict left to right evaluation order can be confusing when applied to mathematical expressions. In the Spacewar! code snippet below, the parentheses make it more clear that the addition happens first:

```
newVelocity ← (ai + ag) * t + velocity
```

Example 2.3: Ship velocity

To send multiple messages to the same receiver a *cascade* can be used to state the receiver once, followed by the cascade of messages separated by semicolons. Here is our earlier Example 1.2 code expressed with a cascade:

```
Transcript
    show: 'Hello World!';
    newLine;
    show: 'I am Cuising'
```

Example 2.4: Cascade of messages

Another example of a cascade from the Spacewar! game:

```
aShip
    velocity: 0 @ 0;
    morphPosition: randomCoordinate value @ randomCoordinate value
```

Example 2.5: Stop and teleport spaceship at a random position

Observe the text here is formatted to ease code understanding. It is possible to write the cascade of messages in one line, but it reduces the readability of the code:

```
Transcript show: 'Hello World!'; newLine; show: 'I am Cuising'
```

The `Transcript` class is frequently helpful in presenting useful information when developing an application. An alternative to the *Ctrl-d* (**D**o *it*)

shortcut is `Ctrl-p` (**P***rint it*), which executes the script and prints the result direcly in the Workspace.

In the Example 2.4, we have requested no special result. Selecting the text and typing `Ctrl-p` results in the default, which is to return the object to which a message is sent, in this case the `Transcript`.

2.3 Message to string entities

As previously stated, a `String` is a sequence of `Character` elements, and repersents plain Text (without any formatting).

In Cuis, as in most programming languages, `String` literals are a convenient syntax for creating `String` instances. These are some examples of `String` literals:

```
'Hello World!'.
'Accented letters: cigüeña, camión, déjà, deçà, forêt, coïncidence'.
'Non-latin (Greek) letters: Λορεμ ιπσθμ δολορ σιτ αμετ'.
```

The first example is an instance of `String`. This class is used for String literals if all Characters are in the limited ASCII character set. The following examples contain non-ASCII Characters. For these, an instance of `UnicodeString` is used instead. You usually don't need to care about this: Instances of `String` and `UnicodeString` understand the same messages, so they are interchangeable.

Access to a character in a string is done with the keyword message `#at:` and its index position in the string as the argument. Execute the following examples with the `Ctrl-p` shortcut:

```
'Hello' at: 1 ⇒ $H
'Hello' at: 5 ⇒ $o
```

Observe how a character is prefixed with the "$" symbol.

Caution. The index indicates position, naturally starting from 1, and is valid up to the string length.

```
'Hello' indexOf: $e
⇒ 2
```

To change one character, use the companion two keyword message `#at:put:`. The argument must be noted as a character:

```
'Hello' at: 2 put: $a; yourself
⇒ 'Hallo'
```

Observe the use of the cascade with the `#yourself` message. A cascade sends the following messages to the original receiver, so `#yourself` returns the updated string. Without the cascade, `$a` is returned as the result of the `#at:put:` message.

> *Replace each character of the string 'Hello' to become 'Belle'.*

Exercise 2.1: Hello to Belle

Characters that are part of the ASCII character set can be asked for their ASCII code. Conversely, given an ASCII code, we can ask for the corresponding `Character`:

```
$A asciiValue
⇒ 65
Character asciiValue: 65 + 25
⇒ $Z
```

But Characters that are not part of ASCII don't have an ASCII. In general, it is better to use Unicode code points instead:

```
$A codePoint
⇒ 65
UnicodeCodePoint codePoint: 65 + 25
⇒ $Z
$ϕ codePoint
⇒ 966
UnicodeCodePoint codePoint: 966
⇒ $ϕ
```

Shuffling a string can be fun but not very useful. However, shuffling can apply to any kind of collection, not only to a string, and it will prove to be useful as we will see later:

```
'hello world' shuffled
⇒ 'wod llreohl'
```

Note that the results of each shuffle are different.

Messages naturally compose:

```
'hello world' shuffled asArray
```

```
⇒ #($h $d $l $ "20" $o $w $e $l $l $o $r)
```

An `Array` literal starts with a hash or sharp character, `$#` and parentheses surround the elements of the array. In this case the elements are `Character`s, but they can be instances of any class.

Like `#shuffled`, all collections answer to the message `#sorted`, which answers a sorted collection.

```
'hello world' sorted
⇒ ' dehllloorw'
```

2.4 Messages to number entities

Earlier, we discussed how Cuis-Smalltalk knows about rational numbers. The four arithmetic operations and mathematical functions are implemented with unary and binary messages understood by the rational numbers:

```
(15 / 14) * (21 / 5) ⇒ 9 / 2
(15 / 14) / ( 5 / 21) ⇒ 9 /2
(3 / 4) squared ⇒ 9 / 16
(25 / 4) sqrt ⇒ 5 / 2
```

Write the code to compute the sum of the squares of the inverse of the first four integers.

Exercise 2.2: Sum of the squares

If Cuis-Smalltalk integer division returns a rational number, how do we find the result in decimal? One option is to write a number as a floating point literal, a *Float*. This kind of literal is written with the integer and fractional parts separated by a dot ".":

```
 15.0 / 4 ⇒ 3.75
15 / 4.0 ⇒ 3.75
```

Another option is to convert an integer to a float with the `#asFloat` message. It is very useful when the integer is in a variable:

```
15 asFloat / 4
⇒ 3.75
```

You can also ask for division with truncation to get an integer result using the message #//:

```
15 // 4
⇒ 3
```

The modulo remainder of the Euclidean division is computed with the message #\\:

```
15 \\ 4
⇒ 3
```

Cuis-Smalltalk has arithmetic operations to test if an integer is odd, even, a prime number, or divisible by another. You just send the appropriate unary or keyword message to the number:

```
25 odd ⇒ true
25 even ⇒ false
25 isPrime ⇒ false
23 isPrime ⇒ true
91 isDivisibleBy: 7 ⇒ true
117 isDivisibleBy: 7 ⇒ false
117 isDivisibleBy: 9 ⇒ true
```

Example 2.6: Testing on integer

With specific *keyword messages* you can compute the Least Common Multiple and Greatest Common Divisor. A keyword message is composed of one or more colons ":" to insert one or more arguments:

```
12 lcm: 15 ⇒ 60
12 gcd: 15 ⇒ 3
```

In the Spacewar! game, the central star is the source of a gravity field. According to Newton's law of universal gravitation, any object with a mass – a spaceship or a torpedo in the game – is subjected to the gravitational force. We compute it as a vector to account for both its direction and its magnitude. The game code snippet below shows a (simplified) mixed scalar and vector calculation done with message sending (See Figure 2.4):

```
-10 * self mass * owner starMass / (position r raisedTo: 3) * position
```

Example 2.7: Computing the gravity force vector

2.5 A brief introduction to the system Browser

Smalltalk organizes instance behaviors using classes. A class is an object which holds a set of methods to be executed when one of its instances receives a message that is the name of one of these methods.

The *System Browser*, in short the *Browser*, is a tool to rule all the classes in Cuis-Smalltalk. It is both a tool to explore the classes (system or user ones) and to write new classes and methods.

To access the tool ...World menu → `Open...` → `Browser`...
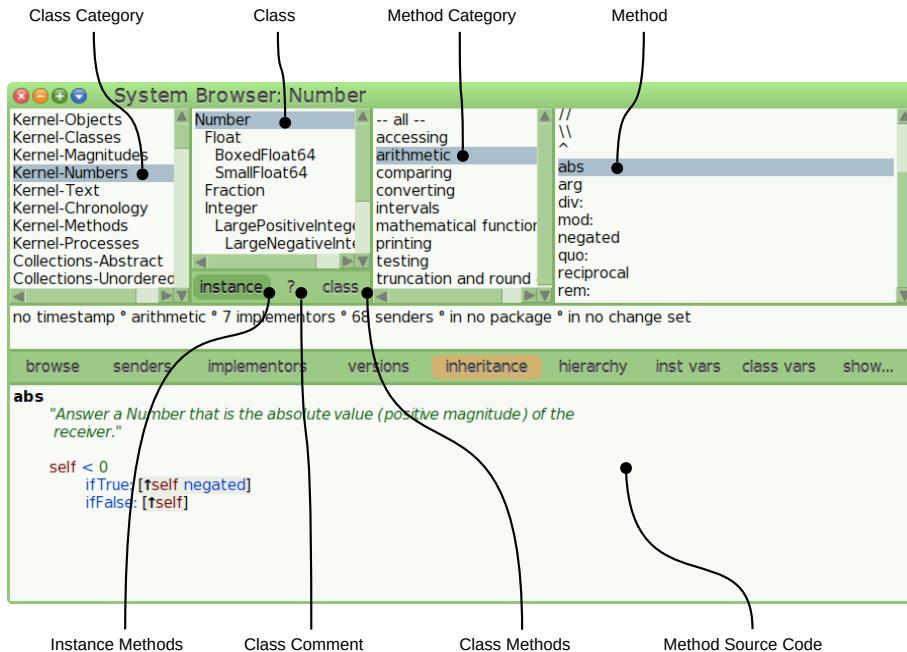
Figure 2.1: The System Browser

At the top left are the *class categories*, groups of classes sharing the same theme. A category can also be used to create a *Package*, which is an organisational element to save code in a file system. In Figure 2.1, the selected class category is **Kernel-Numbers**, a group of classes we already started using. The term **Kernel-** indicates it is part of fundamental categories, but it is only a convention. See the other categories such as **Kernel-Text** and **Kernel-Chronology** related to text and date entities.

Next to the right are the classes in the selected class category. They are nicely presented in a parent-child class hierarchy. When a class is first

selected in this pane, its declaration appears in the large pane below, the `Number` class declaration is:

```
Magnitude subclass: #Number
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Numbers'
```

A few important points in this declaration:

- `Number` is a subclass of `Magnitude`. This means `Number` is a kind of specialized `Magnitude`.

- the declaration itself is Smalltalk code, indeed the message `#subclass:instanceVariableNames:classVariableNames:...` was sent to `Magnitude` to create this class.

- the `subclass:` argument `Number` is prefixed with "`#`". It is a symbol, a kind of unique string. Indeed when declaring the `Number` class, the system does not know about it yet, so it is named as a symbol.

- The `instanceVariableNames:` argument is a string: the instance variables of the class are declared by names separated by a space. There are none in this class definition.

A subclass inherits behaviors from its parent superclass, and so only needs to describe what is different from its superclass. An instance of `Number` adds methods (which define behaviors) unknown to an instance of `Magnitude`. We will explore this in detail as we go forward.

To learn about the purpose of a class, it is good practice to **always** visit the class comment. Often a comment also comes with code examples to learn how to use the object; these code snippets can be selected and executed in place as done from a Workspace. In Figure 2.1, see the `comment` button to read or to edit the comment of the selected class.

To the right of the class panel is the method categories panel. A class may have many methods, so grouping them by category helps other users orient themselves in finding related methods. As a reference, `Number` has more than 100 instance side methods implemented in itself[1]. Clicking the `arithmetic` category directly gives access to related methods in the next and last pane at the right.

---

[1] When considering its parents, the combined method count is more than 300.

A right click on the Class Category pane brings up its context menu. You can select `find class ..` or, as the menu indicates, use *Ctrl-f* (**F**ind), to get a fill-in panel and type part of a class name to match. Try it with `String`.

*How many methods are there in the **arithmetic** method category of the `String` class?*

Exercise 2.3: Count of methods

In the Browser, once a method is selected – as in Figure 2.1, `abs` method – the bottom part shows its source code, ready to be explored or edited. Often, you will find a small comment just after the method name, surrounded by double quotes.

Every object knows its own class and will respond with it when sent the message `#class`.

Tip. In the workspace *Ctrl-b* (**B**rowse) on the class name will open a Browser on the named class:

- In the Workspace, type `2 class` and print with *Ctrl-p*,
- `SmallInteger` is printed and automatically highlighted as the current selection,
- Invoke the Browser on the selected `SmallInteger` class with *Ctrl-b*,
- A new Browser instance opens on the `SmallInteger`, ready to be explored.

2.6 Spacewar! models

2.6.1 First classes

In the last chapter we listed the protagonists of the game. Now, we propose a first implementation of the game model with a set of classes representing the involved entities:

1. the game play ⇒ `SpaceWar` class,
2. a central star ⇒ `CentralStar` class,
3. two space ships ⇒ `SpaceShip` class,

4. torpedoes $\Rightarrow$ `Torpedo` class.

Before defining these classes in Cuis-Smalltalk, we want to create a dedicated class category to group them there.

In any kind of Cuis-Smalltalk window, pressing right-click on a pane will typically bring up a menu of operations you can apply which are specific to that pane.

With the mouse pointer over the class category pane of the Browser – the left-most one – just do:

...right mouse click $\rightarrow$ `add item`... then key in *Spacewar!*

Once our new category is created, the Browser proposes a code template in the method source code pane – the bottom one – to create a new class in the `Spacewar!` category:

```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'
```

We replace the symbol `#NameOfSubclass` with a symbol representing the name of the class we want to create. Let's start with `#SpaceWar`. To save our class, while over the class declaration code do ...right mouse click $\rightarrow$ `Accept`... Cuis-Smalltalk will ask for your initials and name if it hasn't before. Alternatively, you can just do `Ctrl-s` (**S**ave).

Then simply repeat the process for each of `#SpaceShip`, `#CentralStar` and `#Torpedo`. If necessary, to get another class code template, click the class category `Spacewar!`.

When done, your class category should be filled with four classes as in Figure 2.2.
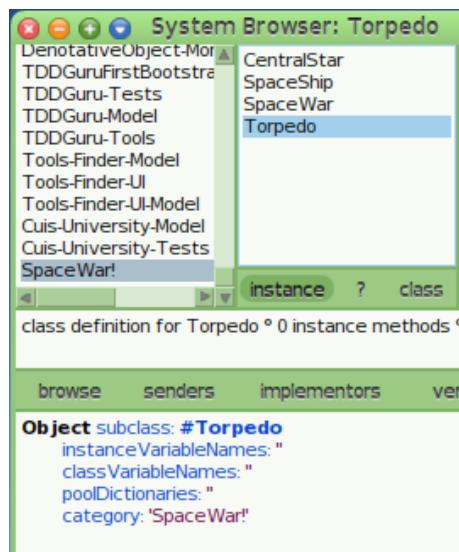
Figure 2.2: Spacewar! class category

2.6.2 Spacewar! package

Another important use case of a class category is to define a package to save code to a file. A package saves the code of the classes held in a given class category and a bit more, but more on that last point later. To create our `Spacewar!` package and save our game code we use the Installed Packages tool.

1. Open the *Installed Packages* tool ...World menu → `Open...` → `Installed Packages...`

2. On the Installed Packages window, do ...click `new` button → Input *Spacewar!* → `Return...`

3. Do ...select `Spacewar!` package → `save` button...
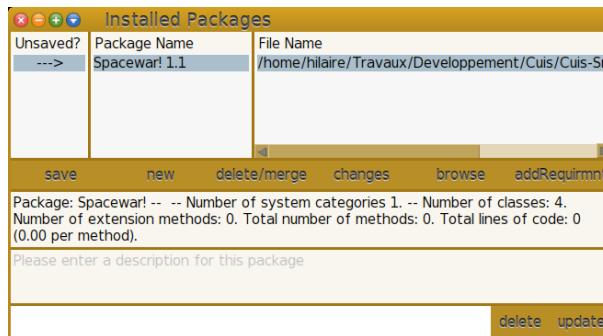
Figure 2.3: Installed Package window

A file `Spacewar!.pck.st` is created alongside the Cuis-Smalltalk image file. To install a package in a fresh Cuis-Smalltalk environment, use the File List tool:

1. Do ...World menu → `Open...` → `File List...`

2. Search for the file `Spacewar!.pck.st` and click the `install package` button

You can also drag and drop the package file from your operating system over to the Squeak window. Upon dropping the file over the window Cuis-Smalltalk will ask you what you want to do with this package. To install it in your enviroment you can simply press `Install package`.

Or, you can open a Workspace, type in `Feature require: 'Spacewar!'` and *Ctrl-d* **D***o it.*

Now, we have created and saved the package. Whenever you start with a fresh Cuis-Smalltalk environment, you can load the game package.

The classes we defined are empty shells with neither state nor behavior. These will be filled in and refactored in the following chapters.

2.6.3 The Newtonian model

For an enjoyable game experience, the player ships must follow Newton's laws of motion. Acceleration, speed and position are computed according to these laws. The ships are subjected to two forces: the acceleration from the gravity pull of the central star and an inner acceleration coming from the ship engines.

Later, we will learn how these equations are easily converted to computer calculations.

Gravity pull

$$\vec{a}_g = \frac{G \cdot m_1 \cdot m_2}{d^2} \cdot \vec{u}$$

$$\vec{a}_g = -\frac{G \cdot m_1 \cdot m_2}{d^2} \cdot \left(\frac{x\,\vec{i} + y\,\vec{j}}{d} \right)$$

$$\vec{a}_g = \frac{-G \cdot m_1 \cdot m_2}{d^3} \cdot \left(x\,\vec{i} + y\,\vec{j} \right)$$

Speed

$$\frac{d\vec{v}}{dt} = \vec{a}_g + \vec{a}_i$$

$$\vec{v} = \left(\vec{a}_g + \vec{a}_i \right) \cdot t + \vec{v}_p$$

$$\vec{v}_p : speed\,at\,previous\,time\,lapse$$

Position

$$\overrightarrow{OM} = \frac{1}{2} \cdot \left(\vec{a}_g + \vec{a}_i \right) \cdot t^2 + \vec{v}_p \cdot t + \overrightarrow{OM}_p$$

$$\overrightarrow{OM}_p : position\,at\,previous\,time\,lapse$$

Legend

$$\alpha : ship\,heading$$

$$\vec{a}_i : internal\,acceleration, \|\vec{a}_i\| = a$$

$$\vec{a}_i = a \left(\cos\alpha\,\vec{i} + \sin\alpha\,\vec{j} \right)$$

$$\|\overrightarrow{OM}\| = d\,,\,distance\,star/ship$$

$$\vec{u} = \frac{-\overrightarrow{OM}}{\|\overrightarrow{OM}\|} = -\frac{x\,\vec{i} + y\,\vec{j}}{d}$$
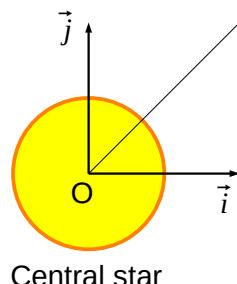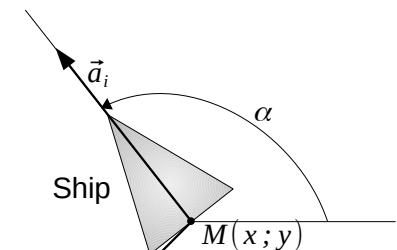

Figure 2.4: Equations of the accelerations, speed and position

3 Class, Model of Communicating Entities

> If I give you something that you can play with and extend, even
> a piece of paper with a paragraph and I say it's not written well,
> rewrite it, that's easier than giving you nothing and say make some-
> thing; you know, giving a blank sheet of paper and starting to write.
> So the lovely part that has proven true for professional program-
> mers as well as kids is when you start with something, an object
> that does something, and then you put many objects like those
> together and have them interact, and then you extend and make
> them behave a little differently, you can take a very incremental
> approach to learning how to control a computer system.
> —*Adele Goldberg*

Cuis-Smalltalk is a pure object oriented programming (OOP) language. All the entities in the language: integers, floats, rational numbers, strings, collections, blocks of code and so forth – every instance usable as a noun in Smalltalk – is an object.

3.1 Understanding Object Oriented Programming

But just what is an object?

At its simplest, an object has two components:

- **Internal state.** This is embodied by variables known only to the object. A variable only visible within the object is called a *private* variable. As a consequence, it is impossible – if the object decides so – to know the internal state of the object from another object.

- **A repertoire of behaviors.** These are the messages an object instance responds to. When the object receives a message it understands, it gets its behavior from a method with that name known by its class or superclass.

The method name is called a *selector* because it is used to select which behavior is invoked. For example, in `'hello' at: 1 put: $B`, the method invoked has the selector `#at:put:` and the arguments `1` and `$B`. All selectors are symbols.

Object instances are created – *instantiated* – following a model or template. This model is known as its *Class*. All instances of a class share the same methods and so react in the same ways.

For example, there is one class `Fraction` but many fractions (1/2, 1/3, 23/17, ...) which all behave the way we expect fractions to behave. The class `Fraction` and the classes it inherits from define this common behavior, as we will now describe.

A given class declares its internal variables – states – and the behavior by implementing the methods. A variable is basically a named box which can hold any object. Each instance variable of a class gets its own box with the common name.

Lets see how the `Fraction` class is declared:

```
Number subclass: #Fraction
   instanceVariableNames: 'numerator denominator'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Numbers'
```

As expected there are two variables – named *instance variables* – to define the `numerator` and `denominator` of a fraction. Each instance of fraction has its own numerator and its own denominator.

From this declaration, we observe there is a hierarchy in the class definition: `Fraction` is a kind of `Number`. This means a fraction inherits the internal state (variables) and behavior (methods) defined in the `Number` class. `Fraction` is called a *subclass* of `Number`, and so naturally we call `Number` a *superclass* of `Fraction`.

A Class specifies the behavior of all of its instances. It is useful to be able to say *this object is like that object, but with these differences*. We do this in Smalltalk by having classes inherit instance state and behavior from their parent Class. This child, or subclass then specifies just the instance state and behavior that is different from its parent, retaining all the unmodified behaviours.

This aspect of object oriented programming is called *inheritance*. In Cuis-Smalltalk, each class inherits from one parent class.

In Smalltalk, we say that each object decides for itself how it responds to a message. This is called *polymorphism*. The same message selector may be sent to objects of different Classes. The *shape* (morph) of the computation is different depending on the specific class of the *many* (poly) possible classes of the object receiving the message.

Different kinds of objects respond to the same `#printString` message in different, but appropriate ways.

We have already met fractions. Those fractions are objects called *instances* of the class `Fraction`. To create an instance we wrote 5 / 4, the mechanism is based on message sending and polymorphism. Let us look into how this works.

The number 5 is an integer receiving the message #/, therefore looking at the method / in the `Integer` class we can see how the fraction is instantiated. See part of this method:

```
/ aNumber
"Refer to the comment in Number / "
| quoRem |
aNumber isInteger ifTrue:
../..
   ifFalse: [↑ (Fraction numerator: self denominator: aNumber) reduced]].
../..
```

From this source code, we learn that in some situations, the method returns a fraction, reduced. We can expect that in some other situation an integer is returned, for example 6 / 2.

In the example, we observe the message #numerator:denominator: is sent to the class `Fraction`, such a message refers to a *class method* understood only by the `Fraction` class. It is expected such a named method returns an instance of a `Fraction`.

Try this out in a workspace:

```
Fraction numerator: 24 denominator: 21
⇒ 24/21
```

Observe how the resulting fraction is not reduced. Whereas it is reduced when instantiated with the #/ message:

```
 24 / 21
⇒ 8/7
```

A class method is often used to create a new instance from a class. In Example 4.7, the message #new is sent to the class `OrderedCollection` to create a new empty collection; `new` is a class method.

In Example 4.8, the #newFrom: message is sent to the class `OrderedCollection` to create a new collection filled with elements from the array given in the argument; `newFrom:` is another class method.

Now observe the hierarchy of the Number class:

```
Number
  Float
    BoxedFloat64
    SmallFloat64
  Fraction
  Integer
    LargePositiveInteger
      LargeNegativeInteger
    SmallInteger
```

`Float`, `Integer` and `Fraction` are direct descendants of the `Number` class. We have already learned about the `#squared` message sent to integer and fraction instances:

```
16 squared ⇒ 256
(2 / 3) squared ⇒ 4/9
```

As the `#squared` message is sent to `Integer` and `Fraction` instances, the associated `squared` method is called an *instance method*. This method is defined in both the `Number` and `Fraction` classes.

Let's examine this method in `Number`:

```
Number>>squared
"Answer the receiver multiplied by itself."
  ↑ self * self
```

In an instance method source code, `self` refers to the object itself, here it is the value of the number. The ↑ (also ^) symbol indicates to *return* the following value `self * self`. One might pronounce ↑ as "return".

Now let's examine this same method in `Fraction`:

```
Fraction>>squared
  ↑ Fraction
      numerator: numerator squared
      denominator: denominator squared
```

Here a new fraction is instantiated with the original instance numerator and denominator being squared. This alternate `squared` method, ensures a fraction instance is returned.

When the message `#squared` is sent to a number, different methods are executed depending on if the number is a fraction or another kind of number. Polymorphism means that the Class of each instance decides how it will repond to a particular message. Here, the `Fraction` class is *overriding* the

`squared` method, defined above in the class hierarchy. If a method is not overridden, an inherited method is invoked to respond to the message.

Still in the `Number` hierarchy, let's examine another example of polymorphism with the `#abs` message:

```
-10 abs ⇒ 10
5.3 abs ⇒ 5.3
(-5 / 3) abs ⇒ 5/3
```

The implementation in `Number` does not need much explanation. There is the `#ifTrue:ifFalse:` we have not yet discussed so far, but the code is quite self-explanatory:

```
Number>>abs
"Answer a Number that is the absolute value (positive magnitude) of the
receiver."
   self < 0
      ifTrue: [↑ self negated]
      ifFalse: [↑ self]
```

This implementation will do just fine for the `Number` subclasses. Nevertheless, there are several classes overriding it for specialized or optimized cases.

For example, regarding large positive integer, `abs` is empty. Indeed, **in the absence of explicitly returned value, the default returned value is the instance itself**, in our situation the `LargePositiveInteger` instance:

```
LargePositiveInteger>>abs
```

The `LargeNegativeInteger` knows it is negative and its absolute value is itself but with its sign reversed, that is `negated`:

```
LargeNegativeInteger>>abs
   ↑ self negated
```

These two overriding methods are more efficient as they avoid unnecessary checks and ifTrue/ifFalse branches. Polymorphism is often used to avoid unnecessary checks and code branches.

> If you select the text `abs` in a Browser or Workspace and right-click to get the context menu, you will find an entry `Implementors of it`. You can select this or use `Ctrl-m` (*i**M**plementors*) to see how various methods for `#abs` use polymorphism to specialize their answer to produce the naturally expected result.

As an object instance is modeled by its class, it is possible to ask any object its class with the `#class` message. Observe carefully the class returned in line 2 and 3:

```
1 class ⇒ SmallInteger
(1/3) class ⇒ Fraction
(6/2) class ⇒ SmallInteger
(1/3) asFloat class  ⇒ SmallFloat64
(1.0/3) class ⇒ SmallFloat64
'Hello' class ⇒ String
('Hello' at: 1) class ⇒ Character
```

Example 3.1: Asking the class of an instance

3.2 Explore OOP from the Browser

In Figure 2.1 of the Browser, below the classes pane, there are three buttons:

- `instance`: to access the **instance methods** of the selected class. These methods apply to each and every instance of the class. Each instance reacts to these.

- `?`: to access to documentation – comment – of the selected class.

- `class`: to access the **class methods** of the selected class. These methods are only accessible from the class itself. Only the class object reacts to class methods.

Below these three buttons, observe the wide text pane, it provides contextual information on the selected item.

Again, Class methods apply to the Class itself. Instance Methods apply for all instances modeled by the class. We saw above that the class `Fraction` has a method `#numerator:denominator:` which is used to get a new instance of a `Fraction`. There is only one `Fraction` class object. Messages like `#squared` and `#abs` are sent to any `Fraction` instance, of which there are many.

Up to now we have attempted to be very careful with definitions, but you know that when we say "a fraction" we mean "an instance of the class `Fraction`". From here our language will be more casual.

> *When the* `Float` *class is selected, what is the information provided by the text pane?*

Exercise 3.1: Float class information

We have spent much time here because it is important to avoid confusion between instance methods and class methods. Let's consider the `Float` class as an example.

Class Methods. In Figure 3.1 the methods listed are class side, in the browser the `class` button is pressed to see this list.


Figure 3.1: Class methods in `Float`

From a Workspace, these methods are called with the message name sent directly to the class:

```
Float e
⇒ 2.718281828459045
Float epsilon
⇒ 2.220446049250313e-16
Float fmax
⇒ 1.7976931348623157e308
```

You have noticed that text typed into the Workspace is colored and highlighted based on what you type. We will discuss this below when we talk about the Smalltalk language, but the idea is to be helpful. If you start to type a word the Cuis Workspace knows about, you can press the `tab` key and get a set of choices for completion of the word. Try typing `Float epsi` and pressing `tab`. You can then press `enter` and should see `Float epsilon`. Click elsewhere on the Workspace to make this menu go away.

Nevertheless, you can not send a class message to an instance of `Float`, it throws an error and opens the red debugger window. Just close the debug window for now to ignore the result.

```
3.14 pi
⇒ MessageNotUnderstood: SmallFloat64>>pi
Float pi e
⇒ MessageNotUnderstood: SmallFloat64>>e
```

Often these class methods are used to access constant values as seen in the previous example or to create a new instance:

```
OrderedCollection new
⇒ Create a new empty ordered collection
Fraction numerator: 1 denominator: 3
⇒ 1/3 "a fraction instance"
Float new
⇒ 0.0
Float readFrom: '001.200'
⇒ 1.2
Integer primesUpTo: 20
⇒ #(2 3 5 7 11 13 17 19)
```

Instance methods. In Figure 3.2, the methods listed are instance side, in the browser the `instance` button is pressed to see this list.
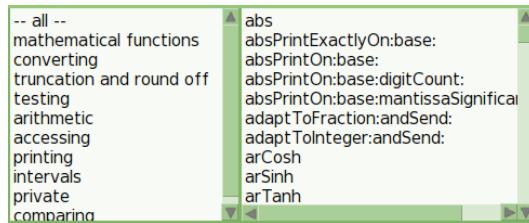
Figure 3.2: Instance methods in `Float`

In a Workspace, these methods are called with the message name sent directly to an instance:

```
-10.12 abs ⇒ 10.12
3.14 cos ⇒ -0.9999987317275395
-10.12 * 2 ⇒ -20.24
```

Instance method messages cannot be sent directly to a class, you need to instantiate an object first:

```
Float cos
⇒ MessageNotUnderstood: Float class>>cos
Fraction squared
⇒ MessageNotUnderstood: Fraction class>>squared
OrderedCollection add: 10
⇒ MessageNotUnderstood: OrderedCollection class>>add:
```

Of course you can mix both class and instance methods, as long as you send the message to the appropriate class or instance:

```
Float pi cos
⇒ -1.0
Float e ln
⇒ 1.0
(Fraction numerator: 4 denominator: 5) squared
⇒ 16/25
OrderedCollection new add: Float pi; add: Float e; yourself
⇒ an OrderedCollection(3.141592653589793 2.718281828459045)
```

Here is another example from Spacewar! mixing class and instance methods. This portion of code updates the orientation of a torpedo according to its velocity vector:

```
self rotation: (velocity y arcTan: velocity x) + Float halfPi
```

Example 3.2: Aligning a torpedo with its velocity direction

With this brief introduction to the system browser, you are now equipped to explore the system classes.

3.3 Cuis system classes

As we noted above, Cuis-Smalltalk is a pure object-oriented environment. This means that every single entitiy you are dealing with is represented as an instance of a class written in Cuis-Smalltalk itself. As a direct consequence, Cuis-Smalltalk is mostly written in itself. This means the entire system is open to you to learn and play with.

What we call system classes are models of fundamental objects. In other programming languages, these would be implemented in that language's standard library.

In a truly open system, there is no real distinction between system classes and user classes, but it will help us to draw a boundary around the most used objects. Let's have a brief introduction to some fundamental Smalltalk classes and their most important methods.

In the upper left pane of the Browser, Categories of classes important to start with are:

- **Kernel-Numbers**. Related to the different number representations and calculations, including mathematics functions, conversion, intervals and even iterations.
- **Kernel-Text**. Related to character and string as collection of characters.
- **Collections-Abstract**, **Collections-Unordered**, **Collections-Sequenceable**, **Collections-Arrayed**. Related to Array, Dictionary, Set, OrderedCollection and many more. This category includes common accessing, enumeration, mathematical functions, and sorting.

3.4 Kernel-Numbers

The top hierarchy `Number` class shows most of the behaviors inherited by the subclasses as `Float`, `Integer` and `Fraction`. The Smalltalk way to learn about a behavior is to point the System Browser toward a top hierarchy class and to explore the method categories.

Let's suppose we want to round a float number. In `Number`, we explore the **Truncation and round off** method category to discover several behaviors. The next thing to do is to test these messages in a Workspace to discover the one we are searching for:

```
1.264 roundTo: 0.1 ⇒ 1.3
1.264 roundTo: 0.01 ⇒ 1.26
1.264 roundUpTo: 0.01 ⇒ 1.27
1.264 roundTo: 0.001 ⇒ 1.264
```

Example 3.3: Rounding numbers, Workspace try out

Number is a very strange place to look for an indexed loop in a given interval. Nevertheless, an interval is defined by start and stop numbers. In the `Number` class, the method category **intervals** reveals related behaviors. These methods work polymorphically with most kinds of number:

```
1 to: 10 do: [:i | Transcript show: 1 / i; space]
⇒ 1 (1/2) (1/3) (1/4) (1/5) (1/6) (1/7) (1/8) (1/9) (1/10)

1 to: 10 by: 2 do: [:i | Transcript show: 1 / i; space]
⇒ 1 (1/3) (1/5) (1/7) (1/9)

1/10 to: 5/3 by: 1/2 do: [:i | Transcript show: i; space]
⇒ (1/10) (3/5) (11/10) (8/5) (1/10) (3/5) (11/10) (8/5)

Float pi to: 5 by: 1/3 do: [:i | Transcript show: i; space]
⇒ 3.141592653589793 3.4749259869231266 3.80825932025646
4.141592653589793 4.474925986923126 4.808259320256459
```

Example 3.4: Interval loops (for-loop)

Now, in the `Integer` class, explore the method category **enumerating**, to find `timesRepeat:`. When a portion of code needs to be executed several times[1], without the need for an index, the `#timesRepeat:` message is sent to an integer. We already saw this variant in a previous section of this chapter. Throwing a 6 face die 5 times can be simulated with an integer:

```
5 timesRepeat: [Transcript show: 6 atRandom; space]
⇒ 1 2 4 6 2
```

Example 3.5: Throwing a die 5 times

Note: Expect a different result each time.

Intervals of numbers can be defined on their own, for future use:

---

[1] More strictly, to be repeated an integer number of times.

```
1 to: 10
⇒ (1 to: 10)

1 to: 10 by: 2
⇒ (1 to: 9 by: 2)
```

Example 3.6: Interval

Intervals work with other kinds of objects such as `Characters`:

```
$d to: $h
⇒ #($d $e $f $g $h)
```

In fact, an interval is an object of its own. It is a sort of collection:

```
(1 to: 10) class
⇒ Interval

(1 to: 10 by: 2) squared
⇒ #(1 9 25 49 81)

(1 to: 10) atRandom
⇒ 4 "different result each time"
```

In Spacewar!, when a ship is destroyed it is teleported to a random position in the square game play area. Intervals are handy to pick random coordinates. In the example below, the variable `randomCoordinate` holds a block of code – called an anonymous function in other languages. It picks a random value in the interval consisting of the gameplay area's left and right extents:

```
randomCoordinate ← [(area left to: area right) atRandom].
aShip
   velocity: 0 @ 0;
   morphPosition: randomCoordinate value @ randomCoordinate value
```

Example 3.7: Teleport ship

> *Compute the cosine values in the interval [0 ; 2PI], each 1/10. Output the result in the transcript.*

Exercise 3.2: Cosine table

Integer numbers are represented in different bases when prefixed with the base and "r". The `r` stands for radix, the base root by which the following number is interpreted. When executing and printing `Ctrl-p` on such a number, it is immediately printed in the decimal base:

```
2r1111 ⇒ 15
16rF ⇒ 15
8r17 ⇒ 15
20rF ⇒ 15
10r15 ⇒ 15
```

Example 3.8: Integer represented by different bases

Writing numbers as Mayans or Babylonians[2]:

```
"The Babylonians"
60r10 ⇒ 60
60r30 ⇒ 180
60r60 ⇒ 360
60r30 + 60r60 ⇒ 540
(60r30 + 60r60) printStringRadix: 60 ⇒ '60r90'

"The Mayans"
20r10 ⇒ 20
20r40 ⇒ 80 "pronounced 4-twenties in some languages"
20r100 ⇒ 400
```

Example 3.9: Counting like the ancients

Because of the nature of a number represented in base 2, shifting its bits left and right is equivalent to multiplying by 2 and dividing by 2:

---

[2] Base 20 and 60 number representations are not exclusive to these civilisations, although they are the most documented use cases.

```
(2r1111 << 1) printStringBase: 2 ⇒ '11110'
2r1111 << 1 ⇒ 30
(2r1111 >> 1) printStringBase: 2 ⇒ '111'
2r1111 >> 1 ⇒ 7
```

Example 3.10: Shifting bits

> *How would you multiply the integer 360 by 1024, without using the multiplication operation?*

Exercise 3.3: Multiply by 1024

Hiatus with decimal numbers

We saw decimal numbers are written with a dot "." to separate the integer and the decimal parts: `1.5`, `1235.021` or `0.5`. The number `0.0000241` is more easily written with the scientific notation `2.41e-5`; it means 2 preceded by 5 zeros or 2 as the fifth digit after the decimal dot.

> Computers encode and store decimal numbers imprecisely. You need to be aware of that when doing computation and equality comparisons. Many systems hide these errors because they are very tiny errors. Cuis-Smalltalk does not hide this inaccuracy. There is good information about this in the class comment of `Float`.

```
0.1 + 0.2 - 0.3
⇒ 5.551115123125783e-17
```

Example 3.11: Computer dyscalculia!

In Example 3.11, the returned value should be zero but it is not the case. The computer returns `5.55e-17`, or `0.0000000000000000555`, it is very close to zero, but there is an error.

Give 3 calculations showing errors compared to the expected results.

Exercise 3.4: Miscellaneous calculation errors with decimal numbers

When accuracy is mandatory use the Rational Numbers representation in Cuis-Smalltalk.

A rational number is written with the division symbol between two integers: do `Ctrl-p` on `5/2` ⇒ `5/2`. Cuis-Smalltalk returns a rational number, it does not compute a decimal.

What happens when executing this code `5/0`*?*

Exercise 3.5: Toward the infinite

Let's come back to our computer's dyscalculia with decimal numbers. When using the rational numbers, the Example 3.11 becomes:

```
(1/10) + (2/10) - (3/10)
⇒ 0
```

Example 3.12: Calculation is correct using fractions!

This time we have the expected result. Under the covers the computer only does the calculations with integer components so no roundoff results. This is a fine example where solving some problems requires a paradigm shift.

> *Return to Exercise 3.4 and use rational numbers to represent decimal numbers. The errors will be resolved.*

Exercise 3.6: Fix the errors

Cuis-Smalltalk knows how to convert a decimal number to a fraction, by sending the message `#asFraction`. We already acknowledged the computer's dyscalculia with decimal numbers, this is why when asking for a fraction representation we have this strange answer. The internal computer representation of `1.3` is not exactly as it seems:

```
(13/10) asFloat
⇒ 1.3

(13/10) asFloat asFraction
⇒ 5854679515581645/45035996273704
```

3.5 Kernel-Text

Notably, this category contains classes `Character`, `String` and `Symbol`. `String` instances are collections of `Character` instances. All these are limited to the small ASCII character set. The corresponding classes that can handle the much larger Unicode character set are `UnicodeCodePoint`, `UnicodeString` and `UnicodeSymbol`. As stated before, the Unicode classes can also handle ASCII, and they are interchangeable. You usually don't need to care about which flavor (ASCII or Unicode) an instance actually is.

Character/UnicodeCodePoint. An individual character is written prefixed with a "$", for example: `$A` or `$φ`. It can be defined with the class side method `codePoint:`. Note, however, that you can ask Character only about valid Characters. It is generally safer to ask UnicodeCodePoint instead:

```
Character codePoint: 65 ⇒ $A
Character codePoint: 966 ⇒ nil
UnicodeCodePoint codePoint: 966 ⇒ $φ
```

There are class side methods for non printable characters: `Character tab`, `Character lf`, etc.

Additionally, `UnicodeCodePoint` defines a `#namedCharactersMap`, that lets you enter many Unicode characters easily, like:

`\leftright` then press spacebar.

As each string is a collection of characters, when iterating a string we can use the `Character` instance methods:

```
'There are 12 apples.' select: [:c |c isDigit].
⇒ '12'
```

Example 3.13: Twelve apples

Modify Example 3.13 to reject the numeric characters.

Exercise 3.7: Select apples

String. `String` is a very large class, it comes with more than 200 methods. It is useful to browse these method categories to see common ways to group methods.

Sometimes you may not see a category related to what you're looking for right away.

If you expect a method selector to start with a specific letter, click-select the `-- all --` method category, then move the cursor over the pane listing the method names. Press this character, e.g. `$f`. This will scroll the method pane to the first method name starting with an "f".

Consider the case where you need to search for a substring, a string within a string: when browsing the `String` class, search for method categories named like **finding...** or **accessing**. There you find a family of `findXXX` methods. Read the comments at the beginning of these methods:

```
findString: subString
   "Answer the index of subString within the receiver, starting at
   start. If the receiver does not contain subString, answer 0."
   ↑ self findString: subString startingAt: 1.
```

Or:

```
findString: key startingAt: start caseSensitive: caseSensitive
   "Answer the index in this String at which the substring key first
```

```
        occurs, at or beyond start.  The match can be case-sensitive or
        not.  If no match is found, zero will be returned."
        ../..
```

Then experiment with the potentially interesting messages in a workspace:

```
'I love apples' findString: 'love' ⇒ 3 "match starts at position 3"
'I love apples' findString: 'hate'
⇒  0 "not found"
'We humans, we all love apples' findString: 'we'
⇒ 12
'We humans, we all love apples'
    findString: 'we'
    startingAt: 1
    caseSensitive: false
⇒ 1
'we humans, we all love apples' findString: 'we'
⇒ 1
'we humans, we all love apples' findString: 'we' startingAt: 2
⇒ 12
```

Following these paths will, most of the time, lead you toward the answer you are looking for.

We want to format a string of the form 'Joe bought XX apples and YY oranges' to the form 'Joe bought 5 apples and 4 oranges'. What message should be used?

Exercise 3.8: Format a string

3.6 Spacewar! States and Behaviors

3.6.1 The game states

After defining the classes involved in the game design, we now define several states of these classes:

- A `SpaceWar` instance representing the game play needs to know about the `centralStar`, the `ships` and the fired `torpedoes`.

- A `CentralStar` has a `mass` state. It is necessary to compute the gravity force applied to a given ship.

- A `SpaceShip` instance knows about its `name`, its `position` coordinates,

its `heading` angle, its `velocity` vector, its `fuel` gauge, its count of the available `torpedoes`, its `mass` and its `acceleration` engine boost.

- A `Torpedo` has `position`, `velocity` and `lifeSpan` states.

We need to explain the mathematical nature of these states, then discuss their object representation in the instance variables of our classes.

> **note** In the following sections, to ease reading we will write "the variable `myVar` is a `String`" instead of the correct but cumbersome "the instance variable `myVar` is a reference to a `String` instance".

SpaceWar

This object is the entry into the game. We want a meaningful class name. Its instance variables are the involved protagonists of the game:

- `centralStar` is the unique `CentralStar` of the game play. We need to know about it to request its mass.
- `ships` is a collection of the two player ships. It is an `Array` instance, its size is fixed to two elements.
- `torpedoes` is a collection of the fired torpedoes in the game play. As this quantity is variable, a dynamic `OrderedCollection` makes sense.

CentralStar

Its unique instance variable, `mass`, is a number, most likely an `Integer`.

SpaceShip

The space ship is the most complex object, some clarifications regarding its variables are needed.

- `name` is a `String`.
- `position` is a 2D screen coordinate, a location. Smalltalk uses the `Point` class to represent such objects. It understands many mathematics operations as operations on vectors; very useful for mechanical calculations.

  A point is easily instantiated with the binary message `#@` send to a number with another number as its argument: `100 @ 200` returns a `Point` instance representing the coordinates (x;y) = (100;200).

  The ship's `position` is regularly recomputed according to the law of the Galilean reference frame. The computation depends on the ship's velocity, it's current engine boost and the gravity pull of the central star.

- `heading` is an angle in radians, the direction where the ship nose is pointing. It is therefore a `Float` number. At game start, the ships are oriented to the top, the `heading` value is then -pi/2 radians; the oy axis is oriented from the top to the bottom of the screen.
- `velocity` is the vector representing the instantaneous speed of the ship. It is a `Point` instance.
- `fuel` is the gauge, as long as it is not zero, the player can ignite the ship's rocket engine to provide acceleration to move around and to counter the central star's gravity pull. It is an integer number.
- `torpedoes` is the quantity of available torpedoes the player can fire. It is an `Integer` number.
- `mass` is an `Integer` representing the ship mass.
- `acceleration` is the intrinsic ship acceleration norm provided when the ship's rockets are ignited. It is therefore an `Integer` number.

A few words regarding the euclidean coordinates: the origin of our orthonormal frame is the central star, its first vector is oriented toward the right of the screen, and the second one towards the top of the screen. This choice eases the computation of the ship's acceleration, velocity and position. More on this below.

Torpedo

A torpedo is launched or "fired" from a ship with an initial velocity related to the ship velocity. Once the torpedo life span counter reaches zero, it self destructs.

- `position` is a 2D screen coordinate, a `Point` instance. Unlike the ship it does not accelerate based on the gravity pull of the central star. Indeed, a torpedo does not come with a mass state. For our purposes it is essentially zero. Its position over time only depends on the torpedo velocity and its initial acceleration.
- `heading` is an angle in radians, the direction where the torpedo nose is pointing. Its value matches the ship heading when fired, it is therefore a `Float` number too.
- `velocity` is a vector representing the instantaneous speed of the torpedo. It is constant over the torpedo lifespan. Again velocity is kept as a `Point` instance.
- `lifeSpan` is an integer number counter, when it reaches zero the torpedo self-destructs.

3.6.2 Instance variables

In the previous chapter, we explained how to define the four classes `SpaceWar`, `CentralStar`, `SpaceShip` and `Torpedo`. In this section, we will add to the these definitions the instance variables – states – discussed above.

To add the variables to the `Torpedo` class, from the Browser, select this class. Next, add the variable names to the `instanceVariableNames:` keyword, separated by one space character. Finally, save the updated class definition with *Ctrl-s* shortcut:

```
Object subclass: #Torpedo
    instanceVariableNames: 'position heading velocity lifeSpan'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'
```

Example 3.14: `Torpedo` class with its instance variables

> *Add the instance variables we discussed earlier to the* `SpaceWar`, `CentralStar` *and* `SpaceShip` *classes.*

Exercise 3.9: Instance variables of the Spacewar! protagonists

3.6.3 Behaviors

Some of these states need to be accessed from other entities:

- When initializing a space ship, we want to set its name with a keyword message categorised as a *setter*: `ship name: 'The needle'`.
- To compute the gravity force applied to an object owning a mass, we want to get its value with an unary message categorised as a *getter*: `star mass * ship mass`.

To write these behaviors in the Browser, first select the class then the method category you want – when none, select `-- all --`.

In the code pane below appears a method template:

```
messageSelectorAndArgumentNames
    "comment stating purpose of message"
    | temporary variable names |
    statements
```

Example 3.15: Method template

It describes itself as:

1. **Line 1.** It is a mandatory method name, the same as the message.

2. **Line 2.** An optional comment surrounded by double quote.

3. **Line 3.** An optional list of variables local to the method, surrounded by pipe characters.

4. **Line 4.** A subsequent list of message sendings and assignments.

The getter `mass` on `SpaceShip` is written as:

```
SpaceShip>>mass
    ↑ mass
```

The `SpaceShip>>` part is not valid code and should not be written in the Browser. It is a text convention to inform the reader the subsequent method is from the `SpaceShip` class.

Write the `SpaceShip` *getter messages for its* `position`, `velocity` *and* `mass` *attributes.*

Exercise 3.10: `SpaceShip` getter message

Some instance variables need to be set from another entity, so a *setter* keyword message is necessary. To set the name of a space ship we add the following method:

```
SpaceShip>>name: aString
    name ← aString
```

The ← character is an assignment, it means the `name` instance variable is bound to the `aString` object. To type in this symbol type _ then space, Cuis-Smalltalk will turn it to left arrow symbol. Alternatively write `name := aString`. One might pronounce ← as "gets".

Since `name` is an instance variable, each instance method knows to use the box for the name. The meaning here is that we are placing the value of the `aString` argument into the instance's box called `name`.

Since each instance variable box can hold an object of any class, we like to name the argument to show that we intend that the `name` variable should hold a string, an instance of the `String` class.

> *Ship* `position` *and* `velocity`, *as well as torpedo* `heading` *will need to be set at game start up or when a ship jumps in hyperspace. Write the appropriate setters.*

Exercise 3.11: `SpaceShip` setter messages

Observe how we do not have a setter message for the space ship `mass` attribute. Indeed, it does not make sense to change the mass of a ship from another object. In fact, if we consider both player ships to be of equal mass, we should remove the `mass` variable and edit the `mass` method to return a literal number:

```
SpaceShip>>mass
    ↑ 1
```

Example 3.16: A method returning a constant

Controls

A space ship controlled by the player understands messages to adjust its direction and acceleration[3]:

Direction. The ship's heading is controlled with the `#left` and `#right` messages. The former decrements the `heading` by 0.1 and the latter increments it by 0.1.

> *Write two methods named* `left` *and* `right` *to shift the ship heading of 0.1 accordingly to the indications above.*

Exercise 3.12: Methods to control ship heading

Acceleration. When the `#push` message is sent to the ship, the engines are ignited and an internal acceleration of 10 units of acceleration is applied to the ship. When the `#unpush` message is sent, the acceleration stops.

---

[3] The velocity is a consequence of the accelerations applied to the space ship.

> *Write two methods named* `push` *and* `unpush` *to adjust the ship inner acceleration accordingly to the indications above.*

Exercise 3.13: Methods to control ship acceleration

3.6.4 Initializing

When an instance is created, for example `SpaceShip new`, it is automatically initialized: the message `#initialize` is sent to the newly created object and its matching `initialize` instance side method is called.

The initializing process is useful to set the default values of the instance variables. When we create a new space ship object we want to set its default position, speed, acceleration:

```
SpaceShip>>initialize
    super initialize.
    velocity ← 0 @ 0.
    position ← 100 @ 100.
    acceleration ← 0
```

Example 3.17: Initialize the space ship

In the method Example 3.17, observe the first line `super initialize`. When a message is sent to `super`, it refers to the superclass of the class's method using `super`. So far, the `SpaceShip` parent class is `Object`, therefore the `Object>>initialize` method is called first for initialization.

When created, a space ship is positioned to the top and right of the central star. It has no velocity nor internal acceleration – only the gravity pull of the central star. Its nose points in direction of the top of the game display.

You probably noticed there is no code to initialize the `heading`, something like `heading ← Float halfPi negated` to orient the ship in the direction of the top of the screen. The truth is we don't need the `heading` attribute, this information will be provided by the class `Morph` used later as a parent class of `SpaceShip` and `Torpedo`. At this time, the `heading` variable will be removed and we will define the heading behavior with the appropriate `heading` and `heading:` methods.

Write the method to initialize the central star with 8000 units of mass?

Exercise 3.14: Initialize central star

4 The Collection Way of Life

Simplicity does not precede complexity, but follows it.
—*Alan Perlis*

Since the concept's introduction in the 70s, collections and their associated iterators are important programming elements of Smalltalk. Correctly used, they improve both code compactness and code understanding; two paradigms which may seem antagonistic. Since then, these innovations have percolated into many popular programming languages.

4.1 String – a particular collection

The `String` class also inherits behavior from its ancestor classes. Indeed `String` is a subclass of `CharacterSequence`, itself a subclass of `SequenceCollection` . The direct consequence is that when searching for some specific behavior, you may need to explore the parent classes too. The whole behavior of a class, defined in the class itself and its parents is called its *protocol*.

Again the browser is helpful to explore a class protocol. You have two options:

1. **Explore the protocol.** In the class pane of the browser, do ...select `String` class → right mouse button → `Browse protocol (p)`... Alternatively, use the keyboard shortcut *Ctrl-p*.
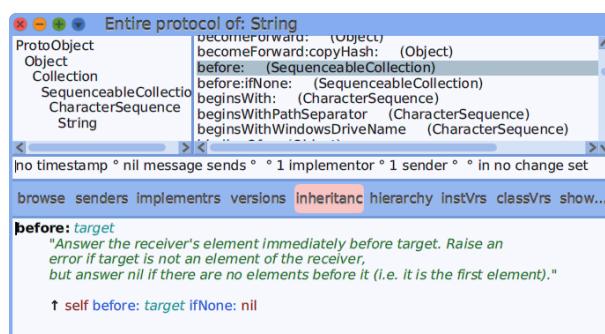
Figure 4.1: Browse `String` protocol

The new window is a protocol browser for the `String` class. At the left, we see a hierarchy of the `String`'s ancestor classes. At the right are the method selectors for strings and, in parenthesis, the class where they are defined. Methods defined in class `String` itself are in bold characters.

Selecting one class there only shows the protocol starting from this class down to the `String` class. If you select `String` in the left panel, you only see methods defined in the `String` class itself.

In Figure 4.1, no specific class is selected, therefore the whole `String` protocol is listed at the right. The method `before:` implemented in `SequenceableCollection` is selected and its source code is displayed on the large bottom pane.

2. **Explore the hierarchy.** In the class pane of the browser, do ...select `String` class → right mouse button → `Browse hierarchy (h)`... Alternatively, use the keyboard shortcut `Ctrl-h` or the button `hierarchy` on the system browser.
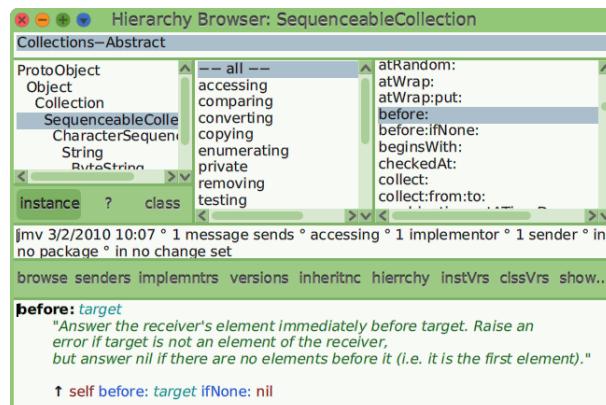
Figure 4.2: Browse the `String` hierarchy

The hierarchy browser is very like the system browser with only two differences:

- At the far left, the class categories pane is absent,
- In the classes pane, the hierarchy of `String` is printed. It makes easy to browse `String` parent and child classes.

The hierarchy browser is a general tool for exploration. Unlike the protocol browser, it does not display the entire protocol of a class. No inherited methods are shown, only those defined directly in the selected class. In Figure 4.2, the class `SequenceableCollection` is selected as well as its method `before:`.

The `before:` method extracts from a collection the element before a specified element. When inherited in `String`, those elements are `Character` instances:

```
'1 + 3i' before: $i
⇒ $3
```

Practice the tools and resolve the exercise below.

> *Find the appropriate method to transform 'Hello My Friend' into 'My Friend'.*

Exercise 4.1: Cut a string

Beware, some messages in the `String` protocol may obviously not work. Observe below, the error is thrown on a `Character` instance:

```
'Hello My Friend' cos
⇒ MessageNotUnderstood: Character>>cos
```

If you look at implementors of `cos`, you can find that `Collection` expects to apply `cos` to each member of a collection, hence a character is asked for its cosine.

Symbol. A symbol is very like a string but it is unique and never duplicated. Two references to `'hello'` might be to two or only one object depending computational history. Two references to `#hello` are guarenteed to always refer to the same object.

Symbols got their name because they are used as *symbolic constants*. You already observed how in the book we wrote message selectors as a symbol. We use symbols because each message name must uniquely index the code for a method. You will use a symbol when you need to name something uniquely.

In the example below, observe the same behavior with string:

```
'hello' == 'hello' copy
⇒ false
#hello  == #hello copy
⇒ true
```

Now you know. A symbol can't be duplicated nor changed.

> In Cuis-Smalltalk, strings with characters not part of the ASCII table are usually instances of `UnicodeString`. In the same way, you may get an instance of `UnicodeSymbol` and not `Symbol`, or `UnicodeCodePoint` and not `Character`. You usually don't need to care about this. The ASCII and Unicode classes provide the same services.

Symbols can contain space characters:

```
'hello my friend' asSymbol
⇒ #'hello my friend'
```

`Symbol` is a subclass of `String` and much of its behavior is inherited. As we learn about `String`s we are also learning quite a bit about symbols.

Note that many `String` methods are defined to return strings.

```
'hello my friend' class.
⇒ String
#'hello my friend' class.
⇒ Symbol
#'hello my friend' asCamelCase
⇒ 'helloMyFriend'
#'hello my friend' asCamelCase asSymbol
⇒ #helloMyFriend
```

4.2 Fun with variables

How a variable can be fun? With Cuis-Smalltalk, a variable is the name of a box that holds a value – an object, that's it!

A variable can hold a value of any class. The value is strongly typed (we can always determine its Class), but the variable (box) is not restricted to holding a value of a single type.

One important direct consequence is that the *type* of a variable – i.e. the class of the referenced object – can change over time. Observe this example:

```
| a |
a ← 1 / 3.
a class
⇒ Fraction
a ← a + (2 / 3)
⇒ 1
a class
⇒ SmallInteger
```

The initial value of the variable `a` was a `Fraction` instance, after some calculation it ends as a `SmallInteger` instance.

In fact to be honest, there is no such things as type, it is only referenced objects which can *mutate* over time into other kind of object: a metal body structure to which you add two wheels may become a bicycle, or a car if you add four wheels.

Therefore, to declare a method variable we just name it at the beginning of the script and surround it by pipe characters "|".

A variable always holds a value. Until we place a different value into a variable, the variable holds the `nil` value, an instance of `UndefinedObject`. When we say that a value is *bound* to a variable we mean that the named box now holds that value.

So far we sent messages directly to objects, but we can send message to a variable bound to an object too.

Any object responds to the message `#printString`.

```
| msg |
msg ← 'hello world!'.
Transcript show: msg capitalized printString, ' is a kind of '.
Transcript show: msg class printString; newLine.
msg ← 5.
Transcript show: msg printString, ' is a kind of '.
Transcript show: msg class printString; newLine.
```

> This ease of use has a drawback: when writing code to send a message to a variable bound to an object, the system does not check ahead of time that the object understands the message. Nevertheless there is a procedure to catch this kind of situation when the message is actually sent.

4.3 Fun with collections

A Collection is a grouping of objects. Arrays and Lists are collections. We already know a `String` is a collection; precisely a collection of characters. Many kinds of Collection have similar behaviors.

An `Array` is a fixed size collection, and unlike a string it can contain any kind of literal enclosed in `#( )`:

```
"array of numbers"
#(1 3 5 7 11 1.1)
```

```
"array of mixed literals"
#(1 'friend' $& 'al')
```

An Array is constructed directly using well formed *literal* elements. We will get to the meaning of this last statement when we discuss details of the Smalltalk language.

For now, just note that using non-literal expressions to construct an array will not work as expected:

```
#(1 2/3)
⇒ #(1 2 #/ 3)
```

Indeed, the `$/` is interpreted as a literal symbol and we get basic components of "2 / 3" but this text is not interpreted as a fraction. To get a fraction inserted in the array, you use a *run-time array* or *dynamic array*, whose elements *are* expressions separated by dots and surrounded with { }:

```
{1 . 2/3 . 7.5}
⇒ #(1 2/3 7.5)
```

With an array filled with numbers you can request information and arithmetic operations:

```
#(1 2 3 4) size ⇒ 4
#(1 2 3 4) + 10 ⇒ #(11 12 13 14)
#(1 2 3 4) / 10 ⇒ #(1/10 1/5 3/10 2/5)
```

Mathematical operations work as well:

```
#(1 2 3 4) squared ⇒ #(1 4 9 16)
#(0 30 45 60) degreeCos
⇒ #(1.0 0.8660254037844386
0.7071067811865475 0.49999999999999994)
```

Basic statistical methods can be used directly on array of numbers:

```
#(7.5 3.5 8.9) mean ⇒ 6.633333333333333
#(7.5 3.5 8.9) range ⇒ 5.4
#(7.5 3.5 8.9) min ⇒ 3.5
#(7.5 3.5 8.9) max ⇒ 8.9
```

To get an array of natural numbers from 1 to 100, we use the keyword message `#to:`

```
(1 to: 100) asArray
```

```
⇒ #(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95 96 97 98 99 100)
```

In this line of code, the message `#to:` is sent to `1` with the argument `100`. It returns an interval object. The message `#asArray` sent to the interval returns an array.

> *Create an array of integer numbers ranging from -80 to 50.*

Exercise 4.2: Negative integer numbers

The size of an array is fixed, it can not grow. An `OrderedCollection` is a dynamic, ordered collection. It grows when adding element with the `#add:` message:

```
| fibo |
fibo ← OrderedCollection newFrom: #(1 1 2 3).
fibo add: 5;
    add: 8;
    add: 13;
    add: 21.
fibo
⇒ an OrderedCollection(1 1 2 3 5 8 13 21)
```

Example 4.1: Dynamic size collection

Index access to the elements of a collection is done with a variety of messages. The index naturally ranges from 1 to the collection size:

```
fibo at: 1 ⇒ 1
fibo at: 6 ⇒ 5
fibo last ⇒ 21
fibo indexOf: 2 ⇒ 3
fibo at: fibo size ⇒ 21
```

Playing with enumerators

A collection comes with a set of helpful methods named enumerators. Enumerators operate on each element of a collection.

Set operations between two collections are computed with the `#union:`, `#intersection:` and `#difference:` messages.

```
#(1 2 3 4 5) intersection: #(3 4 5 6 7)
⇒ #(3 4 5)
#(1 2 3 4 5) union: #(3 4 5 6 7)
⇒ a Set(5 4 3 2 7 1 6)
#(1 2 3 4 5) difference: #(3 4 5 6 7)
⇒ #(1 2)
```

Example 4.2: Set operations

> *Construct the array of the numbers 1,…,24,76,…,100.*

Exercise 4.3: Hole in a set

Set operations work with any kind of object. Comparing objects deserves its own section.

```
#(1 2 3 'e' 5) intersection: #(3.0 4 6 7 'e')
⇒ #(3 'e')
```

To select the prime numbers from 1 to 100, we use the `#select:` enumerator. This message is sent to a collection, then it will select each element of the collection returning true to a test condition:

```
(1 to: 100) select: [ :n | n isPrime ]
⇒   #(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
73 79 83 89 97)
```

Example 4.3: Select prime numbers between 1 and 100

This example introduces the message `#select:` and block of code, a primordial constituent element of the Cuis-Smalltalk model. A block of code, delimited by square brackets, is a piece of code for later execution(s). Let's explain how this script is evaluated:

- `(1 to: 100)` is evaluated as an interval
- the block of code `[:n | n isPrime]` is instantiated (created)
- the message `#select:` is sent to the interval with the block of code as the argument

- in the `select:` method, for each integer of the interval, the block of code is invoked with its parameter **n** set to the integer value. A *block parameter* starts with a colon, ":", and is an ordinary identifer[1]. Then, each time **n isPrime** evaluates to true, the **n** value is added to a new collection answered when the `select:` method finished testing each element of the collection.

A block of code can be saved in a variable, passed as a parameter, and can be used multiple times.

```
| add2 |
add2 ← [:n| n + 2].
{ add2 value: 2. add2 value: 7 }.
⇒  #(4 9)
```

Enumerators implement tremendously powerful ways to process collections without the need of index. By this we mean that they are simple to get right. We like simple!

To get an idea of how useful enumerators are, take a browse at the `Collection` class in the method category `enumerating`.

Select the odd number between -20 and 45.

Exercise 4.4: Odd integers

You want to know the number of prime numbers under 100. Just send the message `#size` to the answered collection at Example 4.3. The parenthesis are mandatory to ensure `#size` is sent last to the resulting collection:

```
( (1 to: 100) select: [:n | n isPrime] ) size
⇒ 25
```

Example 4.4: Quantity of prime numbers between 1 and 100

For more clarity, we use a variable named `primeNumbers` to remember about the prime numbers list we build:

```
| primeNumbers |
```

---

[1] An *identifier* is just a word that starts in a lower case letter and consists of upper and lower case letters and decimal digits. All variable names are identifiers

```
primeNumbers ← (1 to: 100) select: [:n | n isPrime].
primeNumbers size
```

Modify Example 4.4 to calculate the number of prime numbers between 101 and 200.

Exercise 4.5: Number of prime number between 101 and 200

Build the list of the multiples of 7 below 100.

Exercise 4.6: Multiples of 7

Build a collection of the odd integers in [1 ; 100] which are not prime.

Exercise 4.7: Odd and non prime integers

A sister enumerator to `#select:` is `#collect:`. It returns a new collection of the same size, with each element transformed by a block of code. When searching perfect cubic roots, it is useful to know about some cubes:

```
(1 to: 10) collect: [:n | n cubed]
⇒ #(1 8 27 64 125 216 343 512 729 1000)
```

Example 4.5: Collect cubes

The collected elements can be of a different type. Below, a string is enumerated and integers are collected:

```
'Bonjour' collect: [:c | c asciiValue ]
⇒  #(66 111 110 106 111 117 114)
```

We can shift the ascii value, convert back to character then collect in a new string. It is a simple cipher:

```
'Bonjour' collect: [:c | (c asciiValue + 1) asCharacter ]
⇒ 'Cpokpvs'
```

Example 4.6: Simple cipher

Write the script to decode cipher 'Zpv!bsf!b!cptt', it was encoded with Example 4.6.

Exercise 4.8: Cipher decode

The Caesar's cipher is based on shifting letter to the right in the alphabet order. The method is named after Julius Caesar, who used it in his private correspondence with a shift of 3.

Write a script to collect the alphabet upper letters representing the Caesar's cipher. The expected answers is #($D $E $F $G $H $I $J $K $L $M $N $O $P $Q $R $S $T $U $V $W $X $Y $Z $A $B $C).

Exercise 4.9: Alphabet Caesar's cipher

Once you get the alphabet cipher right, you can encode your first message:

Encode the phrase 'SMALLTALKEXPRESSION'.

Exercise 4.10: Encode with Caesar's cipher

And decode message:

Decode this famous quotation attributed to Julius Caesar 'DO-HDMDFWDHVW'.

Exercise 4.11: Decode with Caesar's cipher

Fun with loops

Collection can be iterated with traditional loops: there is a whole family of *repeat*, *while* and *for* loops.

A simple *for* loop between two integer values is written with the keyword message `#to:do:`, the last argument is a block of code executed for each index:

```
| sequence |
sequence ← OrderedCollection new.
1 to: 10 do: [:k | sequence add: 1 / k].
sequence
⇒ an OrderedCollection(1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)
```

Example 4.7: A *for* loop

A collect writes more concisely, though:

```
(1 to: 10) collect: [:k | 1 / k]
```

To step with a different value than 1, a third numeric argument is inserted:

```
1 to: 10 by: 0.5 do: [:k | sequence add: 1 / k]
```

A repeated loop without index or any collection is written with the `#timesRepeat:` message send to an integer:

```
| fibo |
fibo ← OrderedCollection newFrom: #(1 1).
10 timesRepeat: [
   fibo add: (fibo last + fibo atLast: 2)].
fibo
⇒ an OrderedCollection(1 1 2 3 5 8 13 21 34 55 89 144)
```

Example 4.8: A *repeat* loop

The quotient of consecutive Fibonacci terms converge toward the golden value:

```
fibo pairsDo: [:i :j |
   Transcript show: (j / i ) asFloat ; cr]
⇒ 1.0
⇒ 1.5
⇒ 1.6
⇒ 1.6153846153846154
⇒ 1.6176470588235294
⇒ 1.6179775280898876
```

4.4 Collections detailed

The `Collections-` class categories are the most prolific, there are 7 of them gathering 46 classes.

The category `Collections-Abstract` groups classes which are said to be abstract. An *abstract* class cannot be instantiated, its behavior is declared but not completely implemented. It is the responsibility of its subclasses to implement the missing part of the behavior.

An abstract class is useful to establish a set of polymorphic methods which each of its concrete subclasses are expected to specialize. This captures and communicates our intent.

Observe how the important `do:` method is declared but not implemented:

```
Collection>>do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
self subclassResponsibility
```

Then observe how two different `Collection` subclasses implement it:

```
OrderedCollection>>do: aBlock
firstIndex to: lastIndex do: [ :index |
   aBlock value: (array at: index) ]
```

and:

```
Dictionary>>do: aBlock
super do: [:assoc | aBlock value: assoc value]
```

Two important groups of collections must be distinguished: collection with a fixed size and collection with a variable size.

Collection of fixed size. Such collections are gathered in the category `Collections-Arrayed`. The most notable one is `Array`, its size – the number

of elements it can hold – is set when creating the instance. Once instantiated, you can neither add nor delete elements to an array.

There are different ways to create `Array` instance:

```
array1 ← #(2 'Apple' $@ 4) "create at compile time"
array1b ← {2 . 'Apple' . 2@1 . 1/3 } "created a execution time"
array2 ← Array with: 2 with: 'Apple' with: 2@3 with: 1/3.
array3 ← Array ofSize: 4 "an empty array with a 4 element capacity"
```

Example 4.9: Collection with a fixed size

Array `array1` and `array1b` are bit different. The former one is created and filled with its contents at compile time of the code, the consequence is it can only be filled with literal elements as integer, float, string. The later one is created at execution time of the code, it can be filled with elements instantiated at the execution time as `Fraction` or `Point` instances.

You can access elements with an important variety of messages:

```
array1 first ⇒ 2
array1 second ⇒ 'Apple'
array1 third ⇒ $@
array1 fourth ⇒ 4
array1 last ⇒ 4
array1 at: 2 ⇒ 'Apple'
array2 at: 3 ⇒ 2@3
array2 swap: 2 with: 4 ⇒ #(2 1/3 2@3 'Apple')
array1 at: 2 put: 'Orange'; yourself ⇒ #(2 'Orange' $@ 4)
array1 indexOf: 'Orange ⇒ 2
```

Example 4.10: Collection access to elements

Use the System Browser to discover alternative way to access elements of a collection.

> *What is the appropriate message to access the first 2 elements of the* `array1` *collection?*

Exercise 4.12: Access part of a collection

You can't add or remove an element, though:

```
array1 add: 'Orange'
⇒ Error: 'This message is not appropriate for this object'
array1 remove: 'Apple'
⇒  Error: 'This message is not appropriate for this object'
```

Nevertheless, it is possible to fill at once an array:

Fill every element in `array1` *with 'kiwi' all at once?*

Exercise 4.13: Fill an array

Collection of variable size.   Such collection are gathered in several class categories: `Collections-Unordered`, `Collections-Sequenceable`, etc. They represent the most common collections.

`OrderedCollection` is a notable one. Its elements are ordered: elements are added one after the other in sequence[2]. Its size is variable depending on added or removed elements.

```
coll1 ← {2 . 'Apple' . 2@1 . 1/3 } asOrderedCollection
coll2 ← OrderedCollection with: 2 with: 'Apple' with: 2@1 with: 1/3
coll3 ← OrderedCollection ofSize: 4
```

Example 4.11: Collection with a variable size

The access to elements is identical to an `Array` instance, but dynamic collections allow you to add and remove elements:

```
coll1 add: 'Orange'; yourself
⇒ an OrderedCollection(2 'Apple' 2@1 1/3 'Orange')
coll1 remove: 2@1; yourself
⇒ an OrderedCollection(2 'Apple' 1/3)
```

Example 4.12: Adding, removing element from a dynamic array

---

[2] Of course you can insert an element between two elements. However `LinkList` instance are more efficient for this use case.

How to add 'Orange' after 'Apple' in `coll1`*?*

Exercise 4.14: Add an element after

Set. `Set` is an unordered collection without duplicated elements. The order of the element is not guaranteed, though. Observe how pi is the first element of the set:

```
set ← Set new.
set add: 1; add: Float pi; yourself
⇒ a Set(3.141592653589793 1)
```

Example 4.13: Set collection

Non duplicate are guaranteed at least, even with number of different types. Observe how `1`, `3/3` and `1.0` are considered equal and not duplicated in the set:

```
set ← Set new.
set add: 1; add: Float pi; add: 3/3; add: 1/3; add: 1.0; yourself
⇒ a Set(1/3 3.141592653589793 1)
```

Example 4.14: Set, without duplicates

A very handy way to create a `Set` instance, or any other collection, is to create a dynamic array and convert it with the **#asSet** message:

```
{1 . Float pi . 3/3 . 1/3 . 1.0} asSet
⇒ a Set(3.141592653589793 1/3 1)
```

Example 4.15: Convert dynamic array

Observe the alternate conversion messages:

```
{1 . Float pi . 3/3 . 1/3 . 1.0} asOrderedCollection
⇒ an OrderedCollection(1 3.141592653589793 1 1/3 1.0)

{1 . Float pi . 3/3 . 1/3 . 1.0} asSortedCollection
⇒ a SortedCollection(1/3 1 1 1.0 3.141592653589793)
```

To uniquely collect the divisors list of 30 and 45 (not the common divisors):

```
Set  new
    addAll: #(1 2 3 5 6 10 15 30) ;
    addAll: #(1 3 5 9 15 45) ;
    yourself.
⇒ a Set(5 10 15 1 6 30 45 2 3 9)
```

> *How will you collect the letters in the sentences 'buenos días' and 'bonjour'?*

Exercise 4.15: Letters

Dictionary. A dictionary is a list of associations between a key and an object. Of course a key is an object, but it must respond to equality tests. Most of the time, symbols are used as keys.

To compile a list of colors:

```
| colors |
colors ← Dictionary new.
colors
    add: #red -> Color red;
    add: #blue -> Color blue;
    add: #green -> Color green
```

Example 4.16: Dictionary of colors

There are shorter descriptions:

```
colors ← Dictionary newFrom:
    {#red -> Color red . #blue -> Color blue . #green -> Color green}.
colors ← {#red -> Color red . #blue -> Color blue .
    #green -> Color green} asDictionary
```

You access color by symbols:

```
colors at: #blue
⇒ Color blue
colors at: #blue put: Color blue darker
colors at: #yellow ifAbsentPut: Color yellow
```

```
⇒ association `#yellow -> Colors yellow` added to the dictionary
```

There are different way to access a dictionary contents:

```
colors keys.
⇒ #(#red #green #blue)
colors keyAtValue: Color green
⇒ #green
```

Beware. The classic enumerators iterate the values of the dictionary:

```
colors do: [:value | Transcript show:  value; space ]
⇒ (Color r: 1.000 g: 1.000 b: 0.078) (Color r: 0.898 g: 0.000 b: 0.000)...
```

Sometimes, you really need to iterated the whole key-value association:

```
colors associationsDo: [:assoc |
    Transcript show: assoc key; space; assoc value; cr ]
```

There are other variants to explore by yourself.

With an appropriate enumerator, how will you edit the contents of the `colors` *dictionary to replace its values with a nicely capitalized string representing the name of the color?*

Exercise 4.16: Color by name

There are many more collections to explore. You now know enough to explore and to search by yourself with the System Browser, and to experiment with the Workspace.

4.5 SpaceWar! collections

4.5.1 Instantiate collections

Whenever you need to deal with more than one element of the same nature – instances of the same class – it is a clue to use a collection to hold them. Moreover, when these elements are of fixed quantity, it indicates more precisely you want to use an `Array` instance. An `Array` is a collection of fixed size. It can not grow nor shrink.

When this quantity is variable, you want to use an `OrderedCollection` instance. It is a collection of variable size, it can grow or shrink.

SpaceWar! is a two-players game, there will be always two players and two space ships. We use an `Array` instance to keep reference of each space ship.

Each player can fire several torpedoes; therefore the game play holds zero or more torpedoes – hundreds if we decide so. The torpedoes quantity is variable, we want to use an `OrdredCollection` instance to keep track of them.

In the `SpaceWar` class, we already defined two instance variables `ships` and `torpedoes`. Now, we want an `initializeActors` method to set up the game with the involved actors – central star, ships, etc. Part of this initialization is to create the necessary collections.

See below an incomplete implementation of this method:

```
SpaceWar>>initializeActors
    centralStar ← CentralStar new.
    ../..
    ships first
       position: 200 @ -200;
       color: Color green.
    ships second
       position: -200 @ 200;
       color: Color red
```

Example 4.17: Incomplete game initialization

> *The example above does not show the creation of the* `ships` *and* `torpedoes` *collections. Replace "../.." with lines of code where these collections are instantiated and if necessary populated.*

Exercise 4.17: Collections to hold the ships and torpedoes

4.5.2 Collections in action

The space ship and the torpedo objects are responsible of their internal states. They understand the `#update:` message to recompute their position according to the mechanical laws.

A fired torpedo has a constant velocity, no external forces is applied to it. Its position is linearly updated according to the time elapsed. The `t` parameter in the `#update:` message is this time interval.

```
Torpedo>>update: t
"Update the torpedo position"
   position ← velocity * t + position.
   ../..
```

Example 4.18: Torpedo mechanics

A space ship is put under the strain of the star's gravity pull and the acceleration of its engines. Therefore its velocity and position change accordingly to the mechanical laws of physics.

```
SpaceShip>>update: t
"Update the ship position and velocity"
   | ai ag newVelocity |
   "acceleration vectors"
   ai ← acceleration * self direction.
   ag ← self gravity.
   newVelocity ← (ai + ag) * t + velocity.
   position ← (0.5 * (ai + ag) * t squared) + (velocity * t) + position.
   velocity ← newVelocity.
   ../..
```

Example 4.19: Space ship mechanics

Remember that Smalltalk does not follow the mathematics precedence of arithmetic operators. These are seen as ordinary **binary messages** which are evaluated from the left to the right when there is no parenthesis. For example, in the code fragment `...(velocity * t)...`, the parenthesis are mandatory to get the expected computation.

Observe in this previous method how the direction and the gravity are defined in two specific methods.

The `#direction` message asks for the unit vector representing the nose direction of the spaceship:

```
SpaceShip>>direction
"I am an unit vector representing the nose direction of the mobile"
    ↑ Point rho: 1 theta: self heading
```

Example 4.20: Space ship direction method

The `#gravity` message asks for the gravity vector the space ship is subjected to:

```
SpaceShip>>gravity
"Compute the gravity acceleration vector"
    | position |
    position ← self morphPosition.
    ↑ [-10 * self mass * owner starMass / (position r raisedTo: 3) * position]
      on: Error do: [0 @ 0]
```

Example 4.21: Space ship gravity

The game play is the responsibility of a `SpaceWar` instance. At a regular interval of time, it refreshes the states of the game actors. A `stepAt:` method is called at a regular interval of time determined by the `stepTime` method:

```
SpaceWar>>stepTime
"millisecond"
    ↑ 20

SpaceWar>>stepAt: millisecondSinceLast
    ../..
    ships do: [:each | each unpush].
    ../..
```

Example 4.22: Regular refresh of the game play

In the `stepAt:` method, we intentionally left out the details to update the ship and torpedoe positions. Note: each ship is sent regularly an `#unpush` message to reset its previous `#push` acceleration.

Replace the two lines "../.." with code to update the ships and the torpedoes positions and velocities.

Exercise 4.18: Update all ships and torpedoes

Among other things, the game play handles the collisions between the various protagonists. Enumerators are very handy for this.

Ships are hold in array of size 2, we just iterate it with a `#do:` message and a dedicated block of code:

```
SpaceWar>>collisionsShipsStar
    ships do: [:aShip |
        (aShip morphPosition dist: centralStar morphPosition) < 20 ifTrue: [
            aShip flashWith: Color red.
            self teleport: aShip]
    ]
```

Example 4.23: Collision between the ships and the Sun

5 Control Flow Messaging

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it.
> Geniuses remove it.
> —*Alan Perlis*

Cuis-Smalltalk 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. There are only 6 keywords, and **there is no syntax for control structures** or for declaring new classes. Instead, nearly everything is achieved by sending messages to objects. For instance, instead of an if-then-else control structure, Smalltalk sends messages like `#ifTrue:` to `Boolean` objects. As we already know, new (sub)classes are created by sending a message to their superclass.

5.1 Syntactic elements

Expressions are composed of the following building blocks:

1. six reserved keywords, or *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.

5.2 Pseudo-variables

In Smalltalk, there are 6 reserved keywords, or 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 the `Boolean` classes `True` and `False`.
- `self` always refers to the receiver of the currently executing method.

- **super** also refers to the receiver of the current method, 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 uses **super**.

- **nil** is the undefined object. It is the unique instance of the class **UndefinedObject**. Instance variables, class variables and local variables are initialized to **nil**.

- **thisContext** is a pseudo-variable that represents the top frame of the run-time stack. In other words, it represents the currently executing **MethodContext** or **BlockContext**. **thisContext** is normally not of interest to most programmers, but it is essential for implementing development tools like the Debugger and it is also used to implement exception handling and continuations.

5.3 Method syntax

Whereas expressions may be evaluated anywhere in Cuis-Smalltalk (for example in a workspace, in a debugger, or in a browser), methods are normally defined in the System 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 Cuis-Smalltalk.

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. We are already farmiliar with this from previous examples.

Let's take a another look to the method syntax when control flow is involved – our first explanation was Section 3.6 [Spacewar! States and Behaviors], page 48).

Here is the method **keyStroke:** in the class **SpaceWar**.

```
 SpaceWar>>keyStroke: event
"Check for any keyboard stroke, and take action accordingly"
   | key |
   key ← event keyCharacter.
   event isArrowUp ifTrue: [↑ ships first push].
   event isArrowRight ifTrue: [↑ ships first right].
   event isArrowLeft ifTrue: [↑ ships first left].
   event isArrowDown ifTrue: [↑ ships first fireTorpedo].
   key = $w ifTrue: [↑ ships second push].
   ...
```

Example 5.1: SpaceWar! key stroke

Syntactically, a method consists of:

- the method pattern, containing the name (*i.e.*, **keyStroke:**) and any

arguments. Here `event` is a `KeyboardEvent`,

- comments (these may occur anywhere, but the convention is to put one at the top that explains what the method does),
- declarations of local variables (*i.e.*, `key`),
- and any number of expressions separated by dots; here there are 5.

The evaluation of any expression preceded by a ↑ (typed as ^) will cause the method to exit at that point, returning the value of that expression. A method that terminates without explicitly returning the value of some expression will always return the current value of `self`.

Arguments 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.

As you might suspect from Example 2.2, `Smalltalk allClasses size` just sends the `#allClasses` method to a dictionary named `Smalltalk`. As with any other object, you can inspect this dictionary. You can note a case of self-reference here: the value of `Smalltalk at: #Smalltalk` is `Smalltalk`.

5.4 Block syntax

Blocks provide a mechanism to defer the evaluation of expressions. A block is essentially an anonymous function. A block is evaluated 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 subsequent expression).

```
[ 1 + 2 ] value
⇒ 3
```

Blocks may take parameters, each of which is declared with a leading colon. A vertical bar separates the parameter declaration(s) 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 sent `#value:value:`, 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. (A block with a large number of parameters is often a sign of a design problem.)

Blocks may also declare local variables, which are surrounded by vertical bars, just like local variable declarations in a method. Locals are declared after any arguments:

```
[ :x :y | | z | z ← x + y. z ] value: 1 value: 2
⇒ 3
```

Blocks can refer to variables of the surrounding environment. Blocks are said to "close over" their lexical environment, which is a fancy way to say that they remember and refer to variables in their surrounding lexical context – those apparent in their enclosing text.

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.

Consider the example below to compute the divisors of an integer:

```
 | n m |
n ← 60.
m ← 45.
(1 to: n) select: [:d | n \\ d = 0 ].
"⇒ #(1 2 3 4 5 6 10 12 15 20 30 60)"
(1 to: m) select: [:d | m \\ d = 0]
"⇒ #(1 3 5 9 15 45)"
```

Example 5.2: Compute divisors

The problem with this example is the code duplication in the divisor computation. We can avoid duplication with a dedicated block doing the computation and assigning it to a variable:

How will you rewrite Example 5.2 to avoid code duplication?

Exercise 5.1: Block to compute divisors

The `SpaceWar>>teleport:` method contains a nice example using a block to avoid code duplication to generate random abscissa and ordinate coordinates. Each time a new coordinate is needed, the message `#value` is sent to the block of code:

```
SpaceWar>>teleport: aShip
  "Teleport a ship at a random location"
  | area randomCoordinate |
  aShip resupply.
  area ← self morphLocalBounds insetBy: 20.
  randomCoordinate ← [(area left to: area right) atRandom].
  aShip
    velocity: 0 @ 0;
    morphPosition: randomCoordinate value @ randomCoordinate value
```

Example 5.3: `teleport:` method

5.5 Control flow with block and message

Deciding to send *this* message instead of *that* one is called *control flow* – controlling the flow of a computation. Smalltalk offers no special constructs for control flow. Decision logic is expressed by sending messages to booleans, numbers and collections with blocks as arguments.

Test

Conditionals are expressed by sending one of the messages `#ifTrue:`, `#ifFalse:` or `#ifTrue:ifFalse:` to the result of a boolean expression:

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

The class `Boolean` offers a fascinating insight into how much of the Smalltalk language has been pushed into the class library. `Boolean` is the abstract superclass of the *Singleton* classes `True` and `False`[1].

Most of the behaviour of `Boolean` instances can be understood by considering the method `ifTrue:ifFalse:`, which takes two blocks as arguments:

```
(4 factorial > 20) ifTrue: [ 'bigger' ] ifFalse: [ 'smaller' ]
⇒ 'bigger'
```

---

[1] A singleton class is designed to have only one instance. Each of `True` and `False` classes has one instance, the values `true` and `false`.

The method is abstract in `Boolean`. It is implemented in its concrete subclasses `True` and `False`:

```
True>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑ trueAlternativeBlock value

False>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑ falseAlternativeBlock value
```

Example 5.4: Implementations of ifTrue:ifFalse:

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 evaluates the true alternative, while an instance of False evaluates the false alternative. All the abstract `Boolean` methods are implemented in this way for `True` and `False`. Look at another example:

```
True>>not
    "Negation----answer false since the receiver is true."
    ↑ false
```

Example 5.5: Implementing negation

Booleans offer several useful convenience methods, such as `ifTrue:`, `ifFalse:`, `ifFalse:ifTrue:`. You also have the choice between eager and lazy conjunctions and disjunctions:

```
(1 > 2) & (3 < 4)
⇒ false  "must evaluate both sides"
(1 > 2) and: [ 3 < 4 ]
⇒ false  "only evaluate receiver"
(1 > 2) and: [ (1 / 0) > 0 ]
⇒ false  "argument block is never evaluated, so no exception"
```

In the first example, both `Boolean` subexpressions are evaluated, since `&` takes a `Boolean` argument. In the second and third examples, only the first is evaluated, since `and:` expects a `Block` as its argument. The `Block` is evaluated only if the first argument is true.

> *Try to imagine how* `and:` *and* `or:` *are implemented.*

Exercise 5.2: Implementing `and:` and `or:`

In the Example 5.1 at the beginning of this chapter, there are 4 control flow `#ifTrue:` messages. Each argument is a block of code and when evaluated, it explicitly returns an expression, therefore interrupting the method execution.

In the code fragment of Example 5.6 below, we test if a ship is lost in deep space. It depends on two conditions:

1. the ship is out of the game play area, tested with the `#isInOuterSpace` message,

2. the ship takes the direction of deep space, tested with the `#isGoingOuterSpace` message.

Of course, the condition #2 is only tested when condition #1 is true.

```
"Are we out of screen?
If so we move the mobile to the other corner
and slow it down by a factor of 2"
(self isInOuterSpace and: [self isGoingOuterSpace])
  ifTrue: [
     velocity ← velocity / 2.
     self morphPosition: self morphPosition negated]
```

Example 5.6: Ship lost in space

Loop

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
```

You can check all the alternatives in the `controlling` method category of the class `BlockClosure`.

`#timesRepeat:` offers a simple way to implement a fixed iteration:

```
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 '
```

You can check all the alternatives in the `intervals` method category of the class `Number`.

> If the exit condition of method like `whileTrue:` is never satisfied, you may have implemented an *infinite loop*. Just type *Cmd-period* to get the Debugger.

5.6 Spacewar!'s methods

You are already acquainted to the writing of simple methods for the Spacewar! game. We will write some more and learn how to categorize them.

5.6.1 Initializing the game play

We want to add the `initialize` method to our `SpaceWar` class. Of course we need to use the System Browser: ...World menu → `Open...` → `Browser...`

As a reminder, proceed as follows (if necessary observe Figure 2.1):

1. In the `Class Category` pane at the far left, scroll down to the `Spacewar!` category, then select it.
2. In the `Class` pane, select the class `SpaceWar`.
3. Below, click the `instance` button to expose the instance side methods of the `SpaceWar` class. It is the default behavior of the browser anyway,

so you can skip this step as long as you have not clicked on the `class` button.

4. In the `Method Category` pane, select the category `-- all --`. A method source code template shows up in the pane below:

```
messageSelectorAndArgumentNames
  "comment stating purpose of message"
  | temporary variable names |
  statements
```

The template comes in four lines: the method name, a comment, local variable declaration and statements. You can select all and delete it or edit each line of the template as needed.

In our case, we select it all and replace it with the `SpaceWar>>initialize` source code:

```
SpaceWar>>initialize
 "We want to capture keyboard and mouse events,
 start the game loop(step) and initialize the actors."
 super initialize.
 color ← self defaultColor.
 self setProperty: #'handlesKeyboard' toValue: true.
 self setProperty: #'handlesMouseOver:' toValue: true.
 self startSteppingStepTime: self stepTime.
 self initializeActors
```

Example 5.7: Initialize `SpaceWar`

5. Once edited, save-it with *Ctrl-s* or ...right click → `Accept (s)`...

The newly created method shows up in the `Method` pane. You can get it categorized automatically too: with the mouse, go over the `Method Category` pane ...right click → `categorize all uncategorized (c)`..

In the `SpaceWar` *class, add the* `teleport:` *method as defined in Example 5.3 then categorize it in the* `events` *method category.*

Exercise 5.3: Categorize a method

5.6.2 Space ship controls

In a previous chapter, you wrote as an exercise simple implementation of the [control ship methods], page 53. The definitive control methods of the `SpaceShip` class are rewritten as:

```
 SpaceShip>>push
  "Init an acceleration boost"
  fuel isZero ifTrue: [↑ self].
  fuel ← fuel - 1.
  acceleration ← 50

SpaceShip>>unpush
  "Stop the acceleration boost"
  acceleration ← 0

SpaceShip>>right
  "Rotate the ship to its right"
  self heading: self heading + 0.1

SpaceShip>>left
  "Rotate the ship to its left"
  self heading: self heading - 0.1
```

Example 5.8: Ship controls

Observe the `right` and `left` methods, they are mostly identical to the ones asked in Exercise 3.12. We don't modify directly the `heading` attribute, we use the methods `heading:` and `heading` to read and write this information.

Categorize the control methods in a newly created method category named `control`*.*

Exercise 5.4: Categorize control methods

Control will not be complete without the method to fire a torpedo. It is more complex to correctly initialize a torpedo. This is because a space ship is typically in motion, and in addition its heading and velocity are changing frequently. Therefore the torpedo must be set up according to the current space ship position, heading, and velocity before being fired.

```
SpaceShip>>fireTorpedo
  "Fire a torpedo in the direction of
  the ship heading with its velocity"
  | torpedo |
  torpedoes isZero ifTrue: [ ↑ self].
  torpedoes ← torpedoes - 1.
  torpedo ← Torpedo new.
  torpedo
    position: position + self nose;
    heading: self heading;
    velocity: velocity;
    color: self color muchLighter.
  owner addTorpedo: torpedo
```

Example 5.9: Firing a torpedo from a space ship in motion
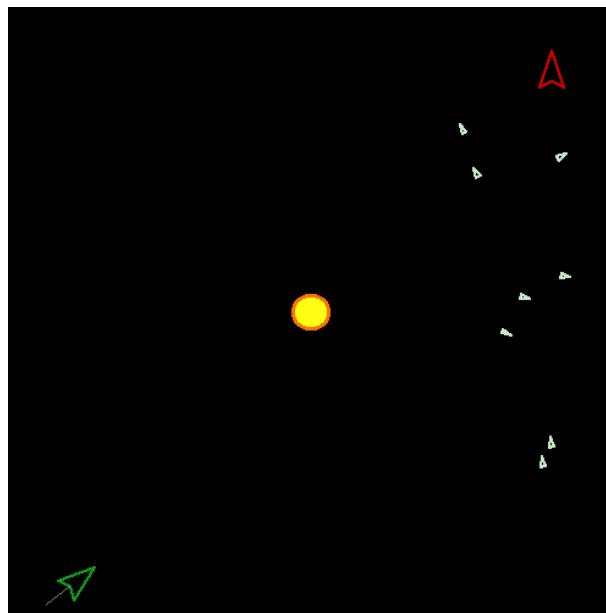
Figure 5.1: Spacewar! torpedoes around

5.6.3 Collisions

In a previous chapter we gave a small taste of the collision detection code between the space ships and the central star. It relies on iterator, block of code and control flow.

However we have other scenarios as ship-ship, torpedo-Sun and torpedo-ship collisions.

> ✎ *How will you write the method to detect the collision between the two ships and take action accordingly? (Adapt from Example 4.23).*

Exercise 5.5: Ships collision

The detection between the two ships and the possible numerous torpedoes required two enumerators with nested blocks of code:

```
SpaceWar>>collisionsShipsTorpedoes
ships do: [:aShip |
  torpedoes do: [:aTorpedo |
    (aShip morphPosition dist: aTorpedo morphPosition) < 15 ifTrue: [
      aShip flashWith: Color red.
      aTorpedo flashWith: Color orange.
      self destroyTorpedo: aTorpedo.
      self teleport: aShip]
  ]
]
```

Example 5.10: Collision between the ships and the torpedoes

The last torpedo-Sun scenario collision is left as an exercise for you.

> ✎ *Write the method to detect the collisions between the torpedoes and the central star and take action accordingly. (Adapt from Example 4.23 and Example 5.10.)*

Exercise 5.6: Collision between the torpedoes and the Sun

6 Visual with Morph

> Morphic is a user interface framework that makes it easy and fun
> to build lively interactive user interfaces.
> —*John Maloney*

What would we expect if we asked for good support for building GUIs in a programming system?

All modern computers (and phones, etc) have high resolution color displays. Any software running on them, that is accessible to a user, needs to show stuff on that Display.

Conventional UI managers (that is, Operating Systems and Web Browsers) started by including only the most basic GUI elements first: basic text editors, buttons, simple lists, scrolling for large content, and (usually) multiple resizeable overlapping windows. Anything else needs to be handled via additional libraries. While there are libraries for handling richer content (D3.js and Matplotlib are examples), the result is not consistent, neither for developers nor for users.

Cuis-Smalltalk takes a different approach, pioneered by Smalltalk-80 and especially Self. We will get into detail in the next chapter, *The Fundamentals of Morph*. For now, let's deal with Morphs directly.

We take the high quality Display for granted, as well as a mouse, finger or other pointing device. And we build on the objective of providing ample possibilities for GUIs both in existing, and in novel styles and designs yet to be invented. Additionally, in the usual Smalltalk way, all the framework code is available for study and modification. There are no third party libraries. Only the lowest level code is precompiled, but that still can be overriden or changed.

Therefore every object you see in Cuis-Smalltalk is a `Morph` or is composed of `Morph`s. Basically, a `Morph` is an object with state and behavior that can also depict itself on a computer display screen.

Because Morphs are useful, when you look at class `Morph` in a Hierarchy Browser you will see a large number of methods and many, many subclasses. But the basic ideas are quite simple.

6.1 Installing a Package

This chapter will require you to install the package `UI-Shapes.pck.st`, it is part of the Cuis-Smalltalk-UI project. To fetch this package from the web, you have two options:

1. Grab the file `UI-Shapes.pck.st` and save it in the same folder as your Cuis-Smalltalk image file.

2. In the Cuis-Smalltalk image folder, clone its Git repository:

   ```
   git clone git@github.com:Cuis-Smalltalk/Cuis-Smalltalk-UI.git
   ```

   Along the `UI-Shapes.pck.st` packages there are other ones you can also discover in this repository.

   Once done, in a Workspace execute the code to install the package:

   ```
   Feature require: 'UI-Shapes
   ```

   Your Cuis-Smalltalk environment is now equiped for the next sections of this chapter.

6.2 Ellipse Morph

Let's start with one of the basic morphs, an `EllipseMorph`. You could write `EllipseMorph new openInWorld` and **Do-it**, but we are doing visual things for now, so let's get a World Menu and select from `New Morph... Basic` submenu and drag onto the desktop.

Every time one obtains a morph from a `New Morph...` submenu, one gets a different morph but made to a standard style.
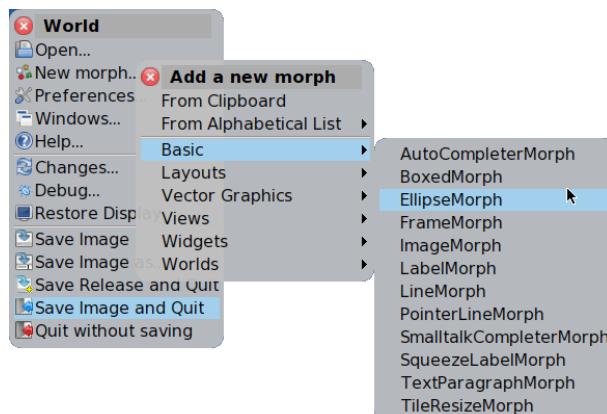
Figure 6.1: Select `EllipseMorph` from a menu

The basic challenge of *user interface design* is to communicate visibility and control. Where am I? What can I do here?

One of the balance points in design is how to eliminate clutter. One useful strategy is to reveal capabilities in context as they are needed.

In the case of Cuis-Smalltalk, you have to know some basics because helpful tools are there but stay out of the way. At any time you can `Right-Click` on the desktop to get the World Menu. You can also `Middle-Click` on any `Morph` to get a *halo* of *construction handles*, which show up as small colored circular icons. If you pause the cursor over one of these, you get a *tool tip*, a temporary text popup who's name should give a clue to its usage.

If you click elsewhere the construction handles leave, but you can get them back at any time with a mouse click.

Figure 6.2: Drag construction handle to change size

Now that you know this, move the yellow lower right handle with tool tip `Change size` via *Click-Drag*. Just hold down the left mouse button while the cursor is over the handle, move the cursor to the right and down, and release the mouse button.
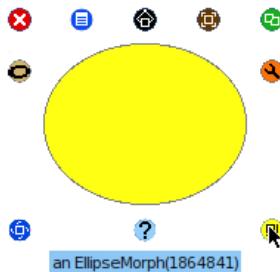
Figure 6.3: A larger ellipse

6.3 Submorph

Morphs can contain other morphs. These interior morphs are called *sub-morph*s of their containing morph. Again, you can do this by writing the software "code", but let's do it directly with a `BoxedMorph`.

First we obtain a `BoxedMorph` from the `New morph...` submenus. The `BoxedMorph` instance displays itself as a rect with a border.
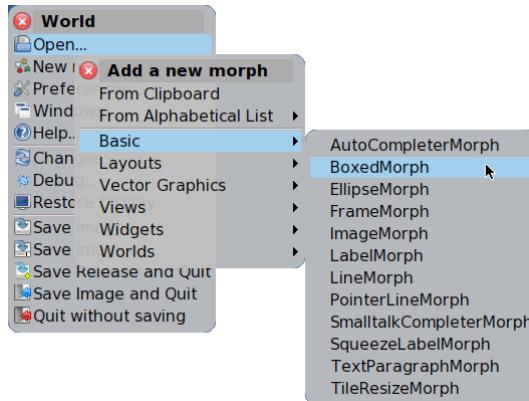
Figure 6.4: Obtain a `BoxedMorph`

Now drag the rect over the ellipse and *Middle-Click* on the rect and click on the blue construction handle to gets the rect's *Morph Menu*. Use the menu selection `embed into` and select the ellipse as its new parent.
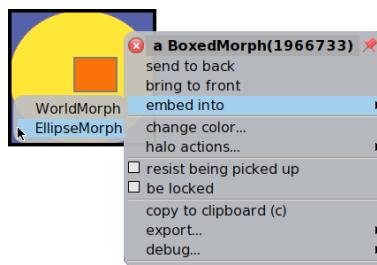
Figure 6.5: Make the rect a submorph of the ellipse

Now when you click-drag the ellipse, or use the `Pick up` or `Move` construction handles, the rect is just a decoration for the ellipse.

Indeed, the rect seems to have fused into the ellipse. Using the mouse where the rect shows itself is just using the mouse on the ellipse. This rect does not have many interesting behaviors.

Let's add a behavior to just this one `BoxedMorph`.

6.4 A brief introduction to Inspectors

To get the construction halo for an interior morph, just `Middle-Click` multiple times to "drill down" through the submorph hierarchy.
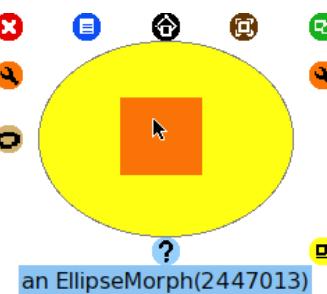
Figure 6.6: Middle-Click for construction handles
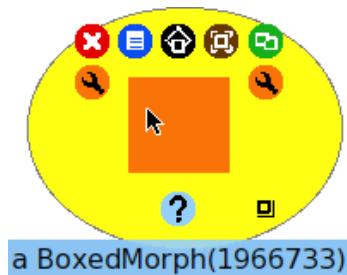
Figure 6.7: Middle-Click again to descend into submorphs

There is an orange handle on the right, just under the green `Duplicate` handle. *Left-Click* this to get the `Debug` menu. Use this menu to get an *Inspector* for the rect.

Observe Figure 6.8, on the left we have a pane for self, all inst vars, and the individual instance variables. Clicking to select "all inst vars" and the values pane on the right shows that the owner of the rect is the ellipse and rect currently has no submorphs.

The lower pane is a Smalltalk code editor, basically a workspace, where `self` is bound to the object we are inspecting.

Inspectors work for every object by the way, not just morphs.
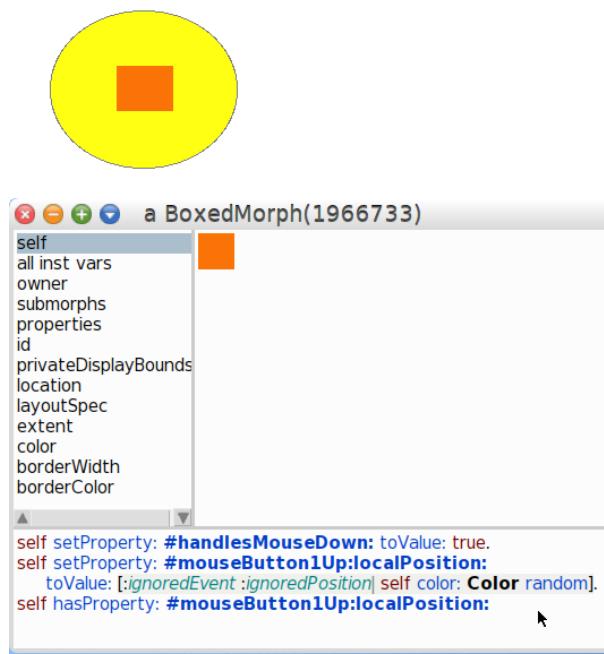
Figure 6.8: Add instance specific behavior

To add a behavior to all instances of a class, we create an instance method. Here we are going to create a behavior for "just this one `BoxedMorph` instance".

In addition to instance variables, a morph can have any number of named *properties* which can be different for each morph.

We add two properties here:

```
self setProperty: #handlesMouseDown: toValue: true.
self setProperty: #mouseButton1Up:localPosition:
    toValue: [:ignoredEvent :ignoredPosition| self color: Color random]
```

Example 6.1: Edit the behavior of this morph from its Inspector

These properties are special to the user interface. You can find methods with these names in the `Morph` class to see what they do.

After selecting the text and **Do-it**, each time you *Left-Click* on the rect it changes color!

Note that you can no longer move the ellipse by mouse-down on the rect, because the rect now takes the mouse event. You have to mouse-down on the ellipse. More on this below.

One quick note on `Move` versus `Pick up`. `Move` moves a submorph "within" its parent. `Pick up` grabs a morph "out" of its parent.
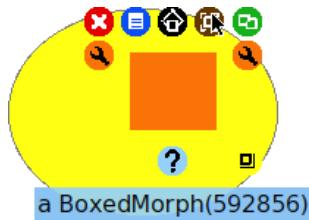
Figure 6.9: Move submorph within its parent
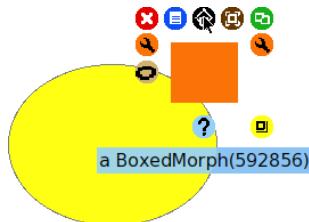
Figure 6.10: Pick a submorph out of its parent

Before we go on, let's use an inspector on the ellipse to change values of a couple of its instance variables.
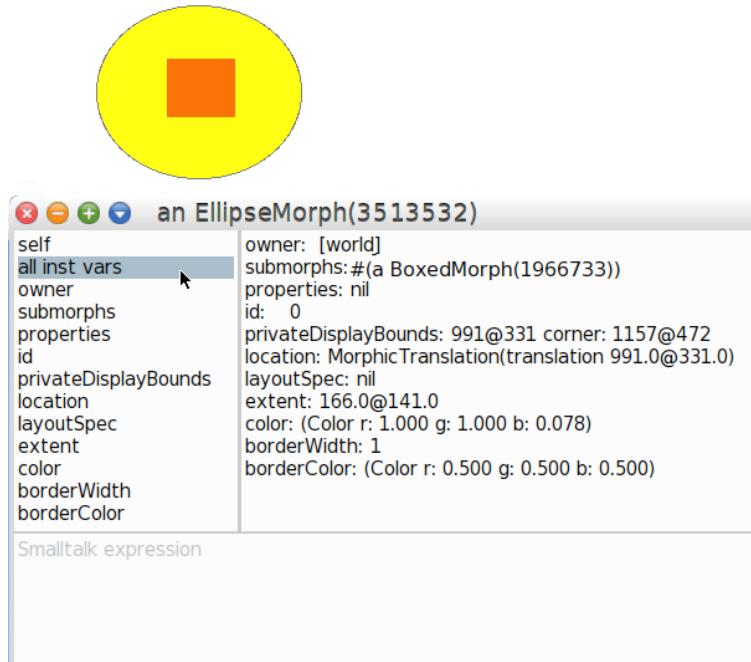
Figure 6.11: Inspect instance variables of the ellipse

Observe Figure 6.12 and Example 6.2. In the lower pane of the inspector, code can be executed in the context of the inspected object. `self` refers to the instance. Here the pane contains code to set the borderWidth and the borderColor.

```
self borderWidth: 10.
self borderColor: Color blue
```

Example 6.2: Edit the state of this ellipse from its Inspector
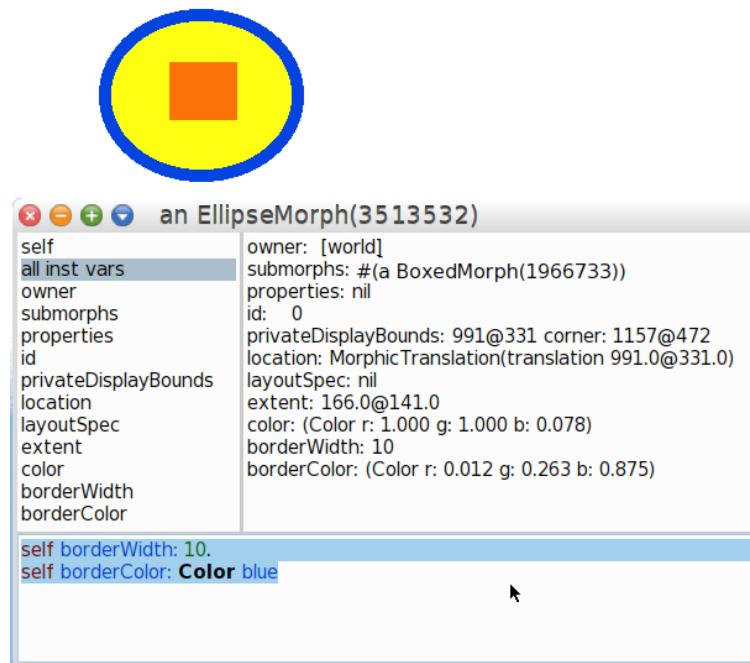
Figure 6.12: Use Inspector to set border color and border width

In the typical case one wants to refine or change behaviors for all instances of a class.

6.5 Building your specialized Morph

Let's make a simple subclass which changes color when *Left-Click*ed. Create a new class just as we did with Spacewar! but subclass `EllipseMorph` with `#ColorClickEllipse`.

```
EllipseMorph subclass: #ColorClickEllipse
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'
```

Save the class definition with *Ctrl-s*.

Right-Click on the `Message Category` pane and select `new category...`. This brings up a number of selections and allows us to

create new ones. Select "event handling testing". Then add the method
`ColorClickEllipse>>handlesMouseDown:`.

```
handlesMouseDown: aMouseButtonEvent
  "Answer that I do handle mouseDown events"
  ↑ true
```

Likewise, add a new catagory "event handing" and add the other method
we need.

```
mouseButton1Up: aMouseButtonEvent localPosition: localEventPosition
  "I ignore the mouseEvent information and change my color."
  self color: Color random
```

Now, you have created a new Morph class and can select a
`ColorClickEllipse` from the World Menu `New Morph..` and try it out.
These are fun to *Left-Click* on. Make as many as you want!
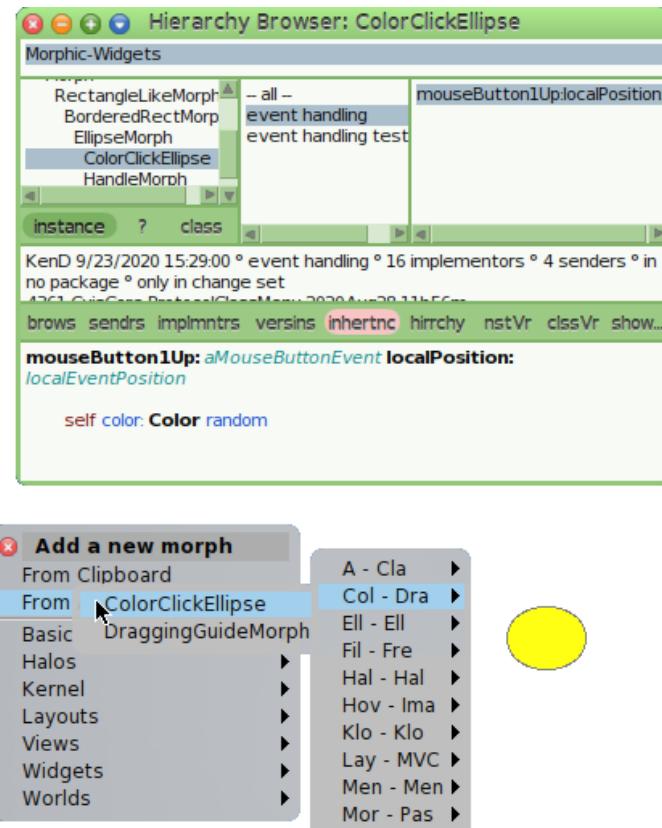
Figure 6.13: Obtain a `ColorClickEllipse`

Now you know how to specialize an individual morph, or make a whole new class of them!

6.6 Spacewar! Morphs

6.6.1 All Morphs

Previously we defined the actors of the game as subclasses of the very general `Object` class (See Example 3.14). However the game play, the central star, the ships and the torpedoes are visual objects, each with a dedicated graphic shape:

- the game play is a simple rectangular area filled with the black color,
- the central star is a fluctuating yellow disk with an orange aura,

- the ships are rotating quadrangles each one painted with a different color,

- a torpedo is a rotating triangle to paint with a different color depending on the firing ship.

Therefore it makes sense to turn these actors into kinds of `Morph`s, the visual entity of Cuis-Smalltalk. To do so, point a System Browser to the class definition of each actor, replace the parent class `Object` by `PlacedMorph`[1], then save the class definition with `Ctrl-s`.

For example, the torpedo class as seen in Example 3.14 is edited as:

```
PlacedMorph subclass: #Torpedo
    instanceVariableNames: 'position heading velocity lifeSpan'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'
```

Moreover, a placed Morph already knows about its position and orientation on screen – it can be dragged and moved in the screen and rotated with the mouse cursor. Therefore the `position` and `heading` instance variables are redundant and should be removed. For now we keep it, it will be removed later when we will know how to replace each of its use cases with its appropriate Morph counterpart.

> *Edit* `SpaceWar`, `CentralStar` *and* `SpaceShip` *to be subclasses of the* `PlacedMorph` *class.*

Exercise 6.1: Make all Morphs

As explained in the previous sections of this chapter, a morph can be embedded within another morph. In Spacewar!, a `SpaceWar` morph instance presenting the game play, it is the *owner* of the central star, space ship and torpedo morphs. Put in other words, the central star, space ships and torpedoes are *submorphs* of a `SpaceWar` morph instance.

The `SpaceWar>>initializeActors` code in Example 4.17 is not complete without adding and positioning the central star and space ships as submorphs of the Spacewar! game play:

---

[1] A `PlacedMorph` is a kind of `Morph` with a supplementary `location` attribute; so it can be instructed to move, to scale and to rotate in the screen.

```
SpaceWar>>initializeActors
   centralStar ← CentralStar new.
   self addMorph: centralStar.
   centralStar morphPosition: 0 @ 0.
   torpedoes ← OrderedCollection new.
   ships ← Array with: SpaceShip new with: SpaceShip new.
   self addAllMorphs: ships.
   ships first
      morphPosition: 200 @ -200;
      color: Color green.
   ships second
      morphPosition: -200 @ 200;
      color: Color red
```

Example 6.3: Complete code to initialize the Spacewar! actors

There are two important messages: **#addMorph:** and **#morphPosition:**. The former asks to the receiver morph to embed its morph argument as a submorph, the latter asks to set the receiver coordinates in its owner's reference frame. From reading the code, you deduce the origin of the owner reference frame is its middle, indeed our central star is in the middle of the game play.

There is a third message not written here, **#morphPosition**, to ask the coordinates of the receiver in its owner's reference frame.

Remember our discussion about the **position** instance variable. Now you clearly understand it is redundant and we remove it from the **SpaceShip** and **Torpedo** definitions. Each time we need to access the position, we just write **self morphPosition** and each time we need to modify the position we write **self morphPosition: newPosition**. More on that later.

6.6.2 The art of refactoring

In our [newtonian model], page 29, we explained the space ships are subjected to the engine acceleration and the gravity pull of the central star. The equations are described in Figure 2.4.

Based on these mathematics, we wrote the **SpaceShip>>update:** method to update the ship position according to the elapsed time – see Example 4.19.

So far in our model, a torpedo is not subjected to the central star's gravity pull nor its engine acceleration. It is supposing its mass is zero which is unlikely. Of course the **Torpedo>>update:** method is simpler than the space ship counter part – see Example 4.18. Nevertheless, it is more accurate and even more fun that the torpedoes are subjected to the gravity

pull[2] and its engine acceleration; an agile space ship pilot could use gravity assist to accelerate a torpedo fired with a path close to the central star.

What are the impacts of these considerations on the torpedo and space ship entities?

1. They will share *common states* as the mass, the position, the heading, the velocity and the acceleration.
2. They will share *common behaviors* as the necessary computations to update the position, direction, gravity pull and velocity.
3. They will have *different states*: a torpedo has a life span state while a space ship has fuel tank capacity and torpedoes stock states.
4. They will have *different behaviors*: a torpedo self destructs when its life span expires, a space ship fires torpedoes and accelerates as long as its fuel tank and its torpedoes count are not zero.

Shared state and behaviors suggest a common class. Unshared states and behaviors suggests specialized subclasses which embody the differences. So let us "factor out" the shared elements of the `SpaceShip` and `Torpedo` classes into a common ancestor class; one more specialized than the `Morph` class they currently share.

Doing such analysis on the computer model of the game is part of the *refactoring* effort to avoid behavior and state duplications while making more obvious the common logic in the entities. The general idea of code refactoring is to rework existing code to make it more elegant, understandable and logical.

To do so, we will introduce a `Mobile` class, a kind of `PlacedMorph` with behaviors specific to a mobile object subjected to accelerations. Its states are the mass, position, heading, velocity and acceleration. Well, as we are discussing refactoring, the mass state does not really makes sense in our game, indeed our mobile's mass is constant. We just need a method returning a literal number and we can then remove the `mass` instance variable. Moreover, as explained previously, a `PlacedMorph` instance already know about its position and heading, so we also remove these two attributes, although there are common behaviors to a Space ship and a torpedo.

It results in this `Mobile` definition:

```
PlacedMorph subclass: #Mobile
    instanceVariableNames: 'velocity acceleration color'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'
```

Example 6.4: Mobile in the game play

---

[2] So a torpedo should come with a mass.

> *What should be the refactored definitions of the* `SpaceShip` *and*
> `Torpedo` *classes?*

Exercise 6.2: Refactoring `SpaceShip` and `Torpedo`

The first behaviors we add to our `Mobile` are its initialization and its
mass:

```
Mobile>>initialize
  super initialize.
  velocity ← 0 @ 0.
  acceleration ← 0

Mobile>>mass
  ↑ 1
```

The next methods to add are the ones relative to the physical calculations.
First, the code to calculate the gravity acceleration:

```
Mobile>>gravity
"Compute the gravity acceleration vector"
  | position |
  position ← self morphPosition.
  ↑ -10 * self mass * owner starMass / (position r raisedTo: 3) * position
```

Example 6.5: Calculate the gravity force

This method deserves a few comments:

- `self morphPosition` returns a `Point` instance, the position of the mo-
  bile in the owner reference frame,

- `owner` is the `SpaceWar` instance representing the game play. It is the
  owner – parent morph – of the mobile. When asking `#starMass`, it
  interrogates its central star mass and return its value:

  ```
  SpaceWar>>starMass
      ↑ centralStar mass
  ```

- In `position r`, the `#r` message asks the radius attribute of a point
  considered in polar coordinates. It is just its length. It is the distance
  between the mobile and the central star.

- `* position` really means multiply the previous scalar value with a `Point`, hence a vector. Thus the returned value is a `Point`, a vector in this context, the gravity vector.

The method to update the mobile position and velocity is mostly the same as in Example 4.19. Of course the `SpaceShip>>update:` and `Torpedo>>update:` version must be both deleted. Below is the complete version with the morph's way of accessing the mobile's position:

```
Mobile>>update: t
"Update the mobile position and velocity"
  | ai ag newVelocity |
  "acceleration vectors"
  ai ← acceleration * self direction.
  ag ← self gravity.
  newVelocity ← (ai + ag) * t + velocity.
  self morphPosition:
     (0.5 * (ai + ag) * t squared)
     + (velocity * t)
     + self morphPosition.
  velocity ← newVelocity.
  "Are we out of screen? If so we move the mobile to the other corner
  and slow it down by a factor of 2"
  (self isInOuterSpace and: [self isGoingOuterSpace]) ifTrue: [
     velocity ← velocity / 2.
     self morphPosition: self morphPosition negated]
```

Example 6.6: `Mobile`'s `update:` method

Now we should add the two methods to detect when a mobile is heading off into deep space.

But first we define the method `morphLocalBounds` in each of our Morph objects. It returns a `Rectangle` instance defined in the Morph coordinates by its origin and extent:

```
SpaceWar>>morphLocalBounds
    ↑ -300 @ -300 extent: 600 @ 600

CentralStar>>morphLocalBounds
    ↑ Rectangle center: 0 @ 0 extent: self morphExtent

Mobile>>morphLocalBounds
    ↑ Rectangle encompassing: self class vertices
```

Example 6.7: Bounds of our Morph objects

```
Mobile>>isInOuterSpace
"Is the mobile located in the outer space? (outside of the game
play area)"
    ↑ (owner morphLocalBounds containsPoint: self morphPosition) not

Mobile>>isGoingOuterSpace
"is the mobile going crazy in the direction of the outer space?"
    ↑ (self morphPosition dotProduct: velocity) > 0
```

Example 6.8: Test when a mobile is "spaced out"

As you see, these test methods are simple and short. When writing Cuis-Smalltalk code, this is something we appreciate a lot and we do not hesitate to cut a long method in several small methods. It improves readability and code reuse. The #containsPoint: message asks the receiver rectangle whether the point in argument is inside its shape.

When a mobile is updated, its position and velocity are updated. However the Mobile subclasses SpaceShip or Torpedo may need additional specific updates. In object oriented programming there is this special mechanism named *overriding* to achieve this.

See the Torpedo>>update: definition:

```
Torpedo>>update: t
   "Update the torpedo position"
   super update: t.
   "orientate the torpedo in its velocity direction, nicer effect
   while inaccurate"
   self heading: (velocity y arcTan: velocity x).
   lifeSpan ← lifeSpan - 1.
   lifeSpan isZero ifTrue: [owner destroyTorpedo: self].
   acceleration > 0 ifTrue: [acceleration ← acceleration - 500]
```

Here the update: method is specialized to the torpedo specific needs. The mechanical calculation done in Mobile>>update: is still used to update the torpedo position and velocity: this is done by super update: t. We already discussed super. In the context of Torpedo>>update: it means search for an update: method in Torpedo's parent class, that class's parent and so on until the method is found, if not a *Message Not Understood* error is signalled.

Among the specific added behaviors, the torpedo orientation along its velocity vector is inaccurate but nice looking. To orient accordingly the torpedo, we adjust its heading with its velocity vector heading.

The life span control, the self-destruction sequence, and the engine acceleration are also handled specifically. When a torpedo is just fired, its engine acceleration is huge then it decreases quickly.

With the System Browser pointed to the `Torpedo>>update:` method, observe the `inheritance` button. It is light green, which indicates the message is sent to `super` too. This is a reminder the method supplies a specialized behavior. The button tool tip explains the color hilight meanings within the method's text. When pressing the `inheritance` button, you browse all implementations of the `update:` method within this inheritance chain.
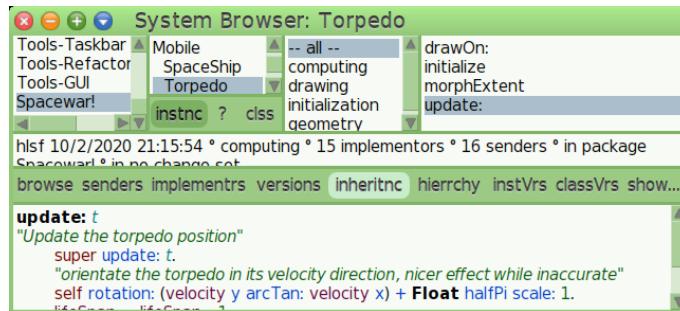
Figure 6.14: Update's inheritance button

We already met an example of overriding when initializing a space ship instance – see Example 3.17. In the context of our class refactoring, the `initialize` overriding spans the whole `Mobile` hierarchy:

```
Mobile>>initialize
    super initialize.
    color ← Color gray.
    velocity ← 0 @ 0.
    acceleration ← 0

SpaceShip>>initialize
    super initialize.
    self resupply

Torpedo>>initialize
    super initialize.
    lifeSpan ← 500.
    acceleration ← 3000
```

Example 6.9: Initialize overriding in the `Mobile` hierarchy

Observe how each class is only responsible of its specific state initialization:

1. **SpaceShip.** Its mechanical states are set with the `super initialize` and then the ship is resupplied with fuel and torpedoes:

   ```
   SpaceShip>>resupply
       fuel ← 500.
       torpedoes ← 10
   ```

2. **Torpedo.** Inherited mechanical states initialized; add self-destroy sequence initialization and acceleration adjusted to mimic the torpedo boost at fire up.

The behaviors specific to each mobile is set with additional methods. The `SpaceShip` comes with its control methods we already described previously in Example 5.8 and Example 5.9, of course there is none for a `Torpedo`.

Another important specific behavior is how each kind of `Mobile` is drawn in the game play, this will be discussed in a next chapter on the fundamentals of Morph.

7 The Fundamentals of Morph

> Simple things should be simple and complex things should be possible.
> —*Alan Kay*

What would we expect if we asked for good support for building GUIs in a programming system?

In Chapter 6 [Visual with Morph], page 90, we started with that same question, and gave an overview of Morphs and their interactive behavior. This chapter deals with how Morphs are built, how to create new Morphs and what rules they follow.

The User Interface framework in Cuis-Smalltalk is called Morphic. Morphic was originally created by Randy Smith and John Maloney as the UI for Self. Later, John Maloney ported it to Smalltalk, to be used as the UI for Squeak.

7.1 Going Vector

For Cuis-Smalltalk, we built Morphic 3, the third design iteration of these ideas, after Self's Morphic 1 and Squeak's Morphic 2. If you already know Morphic in Self or Squeak, most concepts are similar, although with some improvements: Morphic 3 coordinates are not limited to being integer numbers, the apparent size (zoom level) of elements is not tied to pixel density, and all drawing is done with high quality (subpixel) anti aliasing. These enhancements are enabled by the huge advance in hardware resources since Self and Squeak were designed (in the late 80's and late 90's respectively). Additionally, careful design of the framework relieves Morph programmers from much the complexity that was required, especially with regards to geometry.

7.1.1 A first example

Let's start with some examples. What we want is to build our own graphic objects, or Morphs. A Morph class is part of the Morph hierarchy and usually includes a `drawOn:` method for drawing its distinctive appearance. If we forget about computers for a second, and consider drawing with color pens on a sheet of paper, one of the most basic things we can do is to draw straight lines.

So, let's start a System Browser window and build a straight line object:

```
Morph subclass: #LineExampleMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

In method category `drawing` add:

```
LineExampleMorph>>drawOn: aCanvas
    aCanvas strokeWidth: 20 color: Color green do: [
        aCanvas
            moveTo: 100 @ 100;
            lineTo: 400 @ 200 ].
```

Now in a Workspace execute:

```
LineExampleMorph new openInWorld
```

If you get a prompter asking whether to install and activate Vector Graphics support, please answer yes. There it is. You have already built your first `Morph` class.
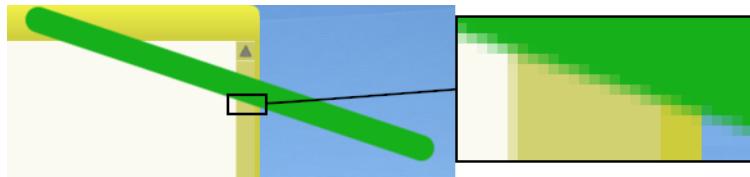
Figure 7.1: Details of our line morph

The code is self evident, the `drawOn:` method takes a `VectorCanvas` instance as an argument. `VectorCanvas` provides many drawing operations for morphs to use. You can play with the various drawing operations and their parameters, and see the result. If you make a mistake, and the `drawOn:` method fails, you'll get a red and yellow error box. After fixing your `drawOn:` method, do ...World menu → `Debug...` → `Start drawing all again..` to get your morph redrawn correctly.

> *How will you modify our line morph so it draws itself as a cross with an extent of 200 pixels?*

Exercise 7.1: Cross morph

7.1.2 Morph you can move

You might have already tried to click and drag on your Line, like you can do with regular windows and most other Morphs. If not, try now. But nothing happens! The reason is that our Morph is fixed in a place in the owner morph (the WorldMorph). It is fixed because `drawOn:` says it should be a line between `100 @ 100` and `400 @ 200`. Moving it around would mean modifying those points. One possible way to do that could be to store those points in instance variables.

But now, we just want to code our morph in the simplest possible way, and still be able to move it around. The solution is to make it subclass of `PlacedMorph`, instead of `Morph`.

To do this, first evaluate the code below to get rid of all LineExample-Morph instances:

```
LineExampleMorph allInstancesDo: [ :m | m delete]
```

Example 7.1: Delete all instances of a given morph

Then, in the System Browser class declaration for LineExampleMorph, type `PlacedMorph` instead of `Morph` and save. Now execute again:

```
LineExampleMorph new openInWorld
```

You will get a line you can grab with the mouse and move it around. `PlacedMorph` adds a new instance variable called `location`. If a morph has a `location`, it can be moved around, by modifying it. The `location` also defines a new local coordinate system. All the coordinates used in the `drawOn:` method are now relative to this new coordinate system. That's why we don't need to modify the `drawOn:` method. `drawOn:` now tells how the morph should be drawn, but not where. The `location` also specifies a possible rotation and scale factor. This means that subinstances of `PlacedMorph` can also be rotated and zoomed.

7.1.3 Filled morph

Let's build another morph, to have more fun.

```
PlacedMorph subclass: #TriangleExampleMorph
    instanceVariableNames: 'borderColor fillColor'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

In method category `initialization` add:

```
TriangleExampleMorph>>initialize
    super initialize.
    borderColor ← Color random alpha: 0.8.
    fillColor ← Color random alpha: 0.6.
```

In the `drawing` method category add:

```
TriangleExampleMorph>>drawOn: aCanvas
    aCanvas strokeWidth: 10 color: borderColor fillColor: fillColor do: [
        aCanvas
            moveTo: 0 @ 100;
            lineTo: 87 @ -50;
            lineTo: -87 @ -50;
            lineTo: 0 @ 100 ].
```

Take a moment to understand that code, to guess what it will do. Now execute:

```
TriangleExampleMorph new openInWorld
```

Do it several times, and move each triangle around. Each new triangle you create has different colors. And these colors are not completely opaque. This means that when you place your triangle over some other morph, you can see through it.
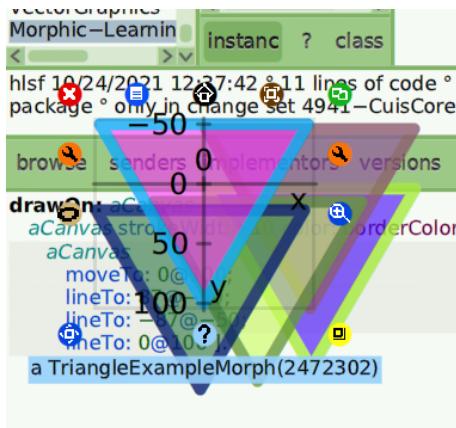
Figure 7.2: A variety of triangle morphs, one decorated with its halo and coordinates system

---

How will you write a movable rectangle morph with an x,y extent of 200 by 100? The rect will be filled with a random translucent color and surrounded by a thin blue line.

Exercise 7.2: Rectangle morph

As we learnt previously, Morphic gives you additional ways to interact with your morphs. With a three button mouse or a wheel mouse, place the mouse pointer (a `HandMorph` instance) over one of your triangles and click with the center button or mouse wheel. If you don't have a three button mouse substitute `Command-click`. You get a constellation of small colored circles around your morph. This is called the morph's *halo*, and each colored circle is a *halo handle*. See Figure 7.2.

At the top left you have the red `Remove` handle. Clicking on it just removes the morph from the morphic world. Hover your hand over each handle, and you'll get a tooltip with its name. Other handles let you `Duplicate` a morph, open a `Menu` with actions on it, `Pick up` (same as dragging it with the the mouse as you did before). The `Move` operation is similar to `Pick up`, but doesn't remove the morph from the current owner. More about that, later.

The `Debug` handle opens a menu from where you can open an Inspector or a Hierarchy Browser to study the morph.

You also have a `Rotate` and `Change scale` handles. Try them! To use them, move your hand to the handle, and then press the mouse button and drag it. As you might have guessed, the rotate handles spins your morph around the center of its surrounding rectangle. The scale handles controls the apparent zoom applied to your morph. Both scale and rotation (and also displacement, as when you move your morph around) are implemented by modifying the inner coordinate system defined by your morph. Displacement, rotation and scale are floating point numbers, and thus not limited to integers.

To change the center of rotation of a Morph, you override the method `rotationCenter` accordingly:

```
RectangleExampleMorph>>rotationCenter
    ↑0 @ 0
```

Observe how our rectangle morph now reacts to the rotation handle. We will learn how to control all this with code and animate our morph.

7.1.4 Animated morph

Let's add two methods to our `TriangleExampleMorph` to make our triangle *alive*:

In the method category `stepping` define:

```
TriangleExampleMorph>>wantsSteps
    ↑ true
```

...and:

```
TriangleExampleMorph>>step
    fillColor ← Color random.
    self redrawNeeded
```

Then create some additional triangles as you did before.

This will make our triangles change color once a second. But more interesting, edit the method:

```
TriangleExampleMorph>>stepTime
    ↑ 100
```

...and:

```
TriangleExampleMorph>>step
    self morphPosition: self morphPosition + (0.4@0).
    self redrawNeeded
```

Now, our morph steps ten times per second, and moves to the right at a speed of four pixels per second. At each step it moves by 0.4 pixels, and not by an integer number of pixels. High quality anti-aliasing drawing means we can actually do that! You can make it step at a speed of four times a second, and move 1 pixel each time, and see how different that looks.

Now try this:

```
TriangleExampleMorph>>step
    self morphPosition: self morphPosition + (0.2@0).
    self rotateBy: 4 degreesToRadians.
    self redrawNeeded
```

It gets even better. First get rid of all instances:

```
TriangleExampleMorph allInstancesDo: [ :m | m delete]
```

And modify these methods:

```
TriangleExampleMorph>>initialize
    super initialize.
    borderColor ← Color random alpha: 0.8.
    fillColor ← Color random alpha: 0.6.
    scaleBy ← 1.1
```

Accept `scaleBy` as a new instance variable of the `TriangleExampleMorph` class.

```
TriangleExampleMorph>>step
    self morphPosition: self morphPosition + (0.2@0).
    self rotateBy: 4 degreesToRadians.
    self scaleBy: scaleBy.
    self scale > 1.2 ifTrue: [scaleBy ← 0.9].
    self scale < 0.2 ifTrue: [scaleBy ← 1.1].
    self redrawNeeded
```

Then create a new triangle:

```
TriangleExampleMorph new openInWorld
```

See that when the triangle is doing its crazy dance, you can still open a halo and interact with it.
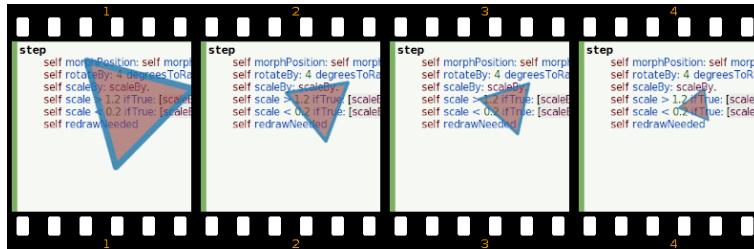
Figure 7.3: Animated morph

7.1.5 Morph in morph

Now, let's try something different. Grab one of your `LineExampleMorph`. With the halo, zoom it until it is about the size of your triangle. Now place the triangle above your line. Open a halo on the triangle, click on the `Menu` handle and select ...`embed into` → `LineExampleMorph`. This makes the triangle a submorph of the line. Now, if you move, scale or rotate the line, the triangle also gets adjusted.

You can open a halo on the triangle. To do this, middle-click twice over it. With the halo on the triangle, you can rotate or zoom it independently of the line. Also note that when you grab the triangle with your hand (not using the halo), you grab the line + triangle composite. You can't just drag the triangle away. For this, you need the triangle's halo. Use its `Move` handle[1] to position it without *getting it out* of the line. Use its `Pick up` handle to take it with the hand and drop it in the world. Now, the triangle is no a longer submorph of the line, and the morphs can be moved, rotated or scaled independently.

But let's try something. Make the triangle submorph of the line again. Now add the following method to category `geometry testing` of the class `LineExampleMorph`:

```
LineExampleMorph>>clipsSubmorphs
    ↑ true
```

The drawing of the triangle gets cut exactly at the bounds of the line. This is most useful for implementing scrolling panes that only make a part of their contents visible, but might have other uses too.

---

[1] By now, it is likely that the triangle has walked quite a bit!
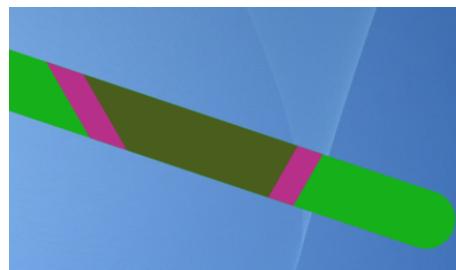
Figure 7.4: An animated and clipped submorph triangle

7.2 A Clock Morph

With all the things we have already learned, we can build a more sophisti-
cated morph. Let's build a `ClockMorph` as see in Figure 7.5.
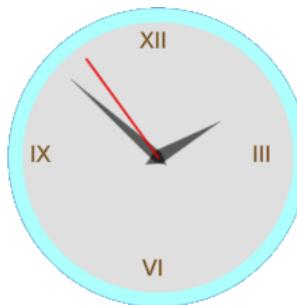
Figure 7.5: A clock morph

Let's create `ClockMorph`, the dial clock :

```
PlacedMorph subclass: #ClockMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

...and its drawing method in the category `drawing`:

```
ClockMorph>>drawOn: aCanvas
    aCanvas
        ellipseCenter: 0@0
        radius: 100
        borderWidth: 10
        borderColor: Color lightCyan
        fillColor: Color veryVeryLightGray.
    aCanvas drawString: 'XII' at: -13 @ -90 font: nil color: Color brown.
    aCanvas drawString: 'III' at: 66 @ -10 font: nil color: Color brown.
    aCanvas drawString: 'VI' at: -11 @ 70 font: nil color: Color brown.
    aCanvas drawString: 'IX' at: -90 @ -10 font: nil color: Color brown
```

Example 7.2: Drawing the clock dial

We create `ClockHourHandMorph`, the hand for the hours:

```
PlacedMorph subclass: #ClockHourHandMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

...and its drawing method in the category `drawing`:

```
ClockHourHandMorph>>drawOn: aCanvas
    aCanvas fillColor: (Color black alpha: 0.6) do: [
        aCanvas
            moveTo: 0 @ 10;
            lineTo: -5 @ 0;
            lineTo: 0 @ -50;
            lineTo: 5 @ 0;
            lineTo: 0 @ 10 ].
```

You can start playing with them. We could use several instances of a single `ClockHandMorph`, or create several classes. Here we chose to do the latter. Note that all the `drawOn:` methods use hardcoded constants for all coordinates. As we have seen before, this is not a limitation. We don't need to write a lot of specialized trigonometric and scaling formulas to build Morphs in Cuis-Smalltalk!

By now, you might imagine what we are doing with all this, but please bear with us while we finish building our clock.

We create `ClockMinuteHandMorph`, the hand for the minutes:

```
PlacedMorph subclass: #ClockMinuteHandMorph
    instanceVariableNames: ''
```

```
classVariableNames: ''
poolDictionaries: ''
category: 'Morphic-Learning'
```

...and its drawing method in the category `drawing`:

```
ClockMinuteHandMorph>>drawOn: aCanvas
    aCanvas fillColor: ((Color black) alpha: 0.6) do: [
        aCanvas
            moveTo: 0 @ 8;
            lineTo: -4 @ 0;
            lineTo: 0 @ -82;
            lineTo: 4 @ 0;
            lineTo: 0 @ 8 ]
```

And finally, the `ClockSecondHandMorph`, the hand for the seconds:

```
PlacedMorph subclass: #ClockSecondHandMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

...and its drawing method in the category `drawing`:

```
ClockSecondHandMorph>>drawOn: aCanvas
    aCanvas strokeWidth: 2.5 color: Color red do: [
        aCanvas
            moveTo: 0 @ 0;
            lineTo: 0 @ -85 ]
```

Now, all that is needed is to put our clock parts together in `ClockMorph`. In its method category `initialization` add its `initialize` method (accept the new names as instance variables):

```
ClockMorph>>initialize
    super initialize.
    self addMorph: (hourHand ← ClockHourHandMorph new).
    self addMorph: (minuteHand ← ClockMinuteHandMorph new).
    self addMorph: (secondHand ← ClockSecondHandMorph new)
```

If you have not already added instance variables for the clock hands, the Cuis IDE will note this and ask what you want to do about it. We want to declare the three missing names as instance variables.
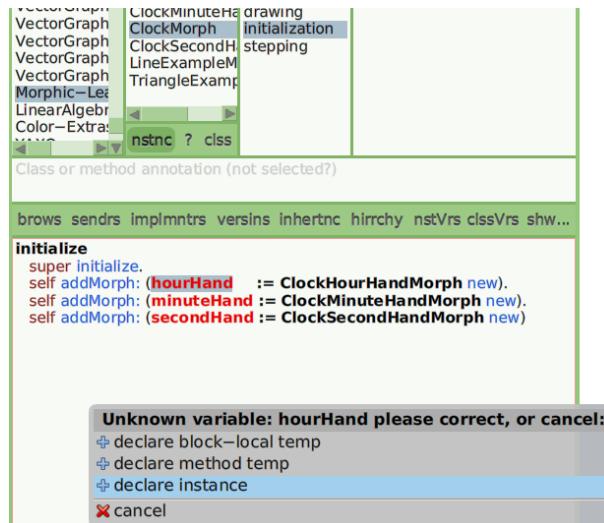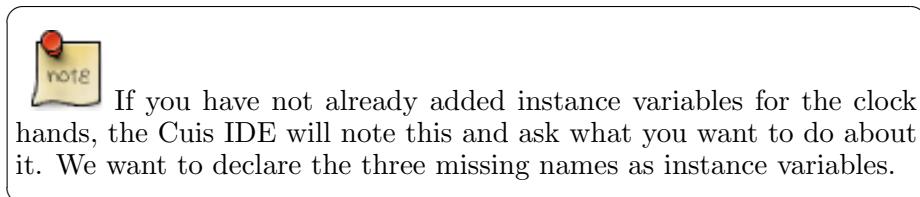
Figure 7.6: Declaring unknown variables as instance variables in current class

Your `ClockMorph` class definition should now be complete!

Figure 7.7: `ClockMorph` with instance variables added

Finally, we animate our clock. In method category `stepping` add the method:

```
ClockMorph>>wantsSteps
    ↑ true
```

...and:

```
ClockMorph>>step
    | time |
    time ← Time now.
    hourHand rotationDegrees: time hour * 30.
    minuteHand rotationDegrees: time minute * 6.
    secondHand rotationDegrees: time second * 6
```

Take a look at how we update the clock hands.

As we said before, any `PlacedMorph` defines a coordinate system for its own `drawOn:` method and also for its submorphs. This new coordinate system might include rotation or reflexion of the axis, and scaling of sizes, but by default they don't. This means that they just translate the origin, by specifying where in the owner point `0 @ 0` is to be located.

The World coordinate system has `0 @ 0` at the top left corner, with X coordinates increasing to the right, and Y coordinates increasing downwards. Positive rotations go clockwise. This is the usual convention in graphics frameworks. Note that this is different from the usual mathematics convention, where Y increases upwards, and positive angles go counterclockwise.

So, how do we update the hands? For example, for the hour hand, one hour means 30 degrees, as 12 hours means 360 degrees or a whole turn. So, we multiply hours by 30 to get degrees. Minute and second hand work in a similar way, but as there are 60 minutes in one hour, and 60 seconds in one minute, we need to multiply them by 6 to get degrees. As rotation is done around the origin, and the clock has set the origin at its center (Example 7.2), there's no need to set the position of the hands. Their `0 @ 0` origin will therefore be at the clock `0 @ 0`, i.e. the center of the clock.
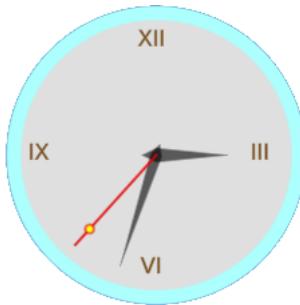
Figure 7.8: A fancy clock morph

> *Look at the clock on Figure 7.8. Don't you think its hand for the seconds decorated with a red and yellow disc is fancy? How will you modify our clock morph to get this result?*

Exercise 7.3: A fancy clock

Create some instances of your clock: `ClockMorph new openInWorld`. You can rotate and zoom. Look at the visual quality of the Roman numerals in the clock face, especially when rotated and zoomed. You don't get this graphics quality on your regular programming environment! You can also extract the parts, or scale each separately. Another fun experiment is to extract the Roman numerals into a separate `ClockFaceMorph`, and make it submorph of the Clock. Then, you can rotate just the face, not the clock, and the clock will show fake time. Try it!

You might have noted two things that seem missing, though: How to compute bounding rectangles for Morphs, and how to detect if a Morph is being hit by the Hand, so you can move it or get a halo. The display rectangle that fully contains a morph is required by the framework to manage the required refresh of Display areas as a result of any change. But you don't need to know this rectangle in order to build your own Morphs. In Cuis-Smalltalk, the framework computes it as needed, and stores it in the `privateDisplayBounds` variable. You don't need to worry about that variable at all.

With respect to detecting if a Morph is being touched by the Hand, or more generally, if some pixel belongs to a Morph, truth is that during the drawing operation of a Morph, the framework indeed knows all the pixels it is affecting. The `drawOn:` method completely specifies the shape of the Morph. Therefore, there is no need to ask the programmer to code the Morph geometry again in a separate method! All that is needed is careful design of the framework itself, to avoid requiring programmers to handle this extra complexity.

The ideas we have outlined in this chapter are the fundamental ones in Morphic, and the framework is implemented in order to support them. Morphs (i.e. interactive graphic objects) are very general and flexible. They are not restricted to a conventional widget library, although such a library (rooted in `BoxedMorph`) is included and used for building all the Smalltalk tools.

The examples we have explored use the `VectorGraphics` framework. It includes `VectorCanvas` and `HybridCanvas` classes. Cuis-Smalltalk also includes the legacy `BitBltCanvas` class inherited from Squeak. `BitBltCanvas` doesn't support the vector graphics drawing operations and doesn't do anti-aliasing or zooming. But it is mature, and it relies on the BitBlt operation that is included in the VM. This means that it offers excellent performance.

To further explore Cuis-Smalltalk' Morphic, evaluate `Feature require: 'SVG'`, and then `SVGMainMorph exampleLion openInWorld` and the other examples there. Also, be sure to try the examples in the class category `Morphic-Examples`, among them execute `Sample10PythagorasTree new openInWorld` and play with the mouse wheel's up and down, left and right directions.

Before we leave this section, here is a two lines change to turn our Cuis quartz clock[2] to an automatic Swiss clock[34]:

```
ClockMorph>>stepTime
    ^ 100 "milliseconds"
ClockMorph>>step
../..
    secondHand rotationDegrees: (time second + (time nanoSecond * 1.0e-9))* 6
```

Try to understand how these changes affect the behavior of the seconds' hand and at which fraction of a second it is rotating.

7.3 Back to Spacewar! Morphs

---

[2] In a quartz clock, the hand for the seconds moves every second.

[3] In an automatic clock, the hand for the seconds moves every fraction of a second. The smaller the fraction, the more premium the clock is.

[4] A Japanese automatic clock will be just fine too.

7.3.1 Central star

Our central star has an extent of `30 @ 30` we need to use in several places in
the code. Therefore it makes sense to define a dedicated method to answer
this value:

```
CentralStar>>morphExtent
    ↑ `30 @ 30`
```

Example 7.3: Central star extent

> An expression surrounded with backticks '`' is evaluated only
> once, when the method is first saved and compiled. This creates a
> compound literal value and improves the performance of the method
> since the expression is not evaluated each time the method is called:
> the pre-built value is used instead.

As you learnt previously, a morph draws itself from its `drawOn:` method.
We draw the star as an ellipse with randomly fluctuating x and y radius:

```
CentralStar>>drawOn: canvas
    | radius |
    radius ← self morphExtent // 2.
    canvas ellipseCenter: 0 @ 0
       radius: (radius x + (2 atRandom − 1)) @ (radius y + (2 atRandom − 1))
       borderWidth: 3
       borderColor: Color orange
       fillColor: Color yellow
```

Example 7.4: A star with a fluctuating size

The star diameters in the x and y directions are fluctuating independently
of 0 to 2 units. The star does not look perfectly round.
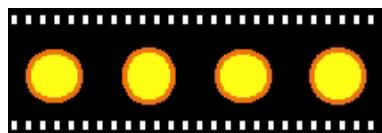
Figure 7.9: A star with a fluctuating size

7.3.2 Space ship

At the game start-up, the nose of the space ship is pointing to the top of the screen as seen in Figure 7.10 and the angle of its direction is therefore -90°, while the angle of its rotation is 0°. Remember the Y ordinate are oriented toward the bottom of the screen.
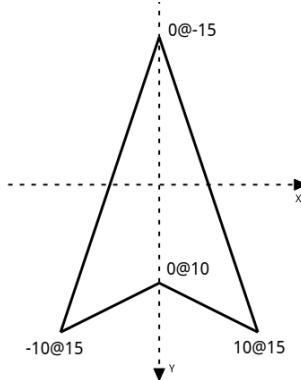
Figure 7.10: Space ship diagram at game start-up

Then its **drawOn:** method is written as:

```
SpaceShip>>drawOn: canvas
    | a b c d |
    a ← 0 @ -15.
    b ← -10 @ 15.
    c ← 0 @ 10.
    d ← 10 @ 15.
    canvas line: a to: b width: 2 color: color.
    canvas line: b to: c width: 2 color: color.
    canvas line: c to: d width: 2 color: color.
    canvas line: d to: a width: 2 color: color.
    "Draw gas exhaust"
    acceleration ifNotZero: [
        canvas line: c to: 0 @ 35 width: 1 color: Color gray]
```

Example 7.5: Space ship drawing

When there is an acceleration from the engine, we draw a small gray line to represent the gas exhaust.

When the user turns the ship, the morph is rotated a bit by adjusting its heading:

```
SpaceShip>>right
"Rotate the ship to its right"
   self heading: self heading + 0.1

SpaceShip>>left
"Rotate the ship to its left"
   self heading: self heading - 0.1
```

But how does this heading affect the rotation of the morph?

Underneath, the `MobileMorph` is equipped with an affine transformation to scale, rotate and translate the coordinates passed as arguments to the drawing messages received by the canvas. Therefore, our `heading` methods are defined to match this internal representation and we use the `rotation:` method from the `PacedMorph` class to rotate appropriately. The `location` attribute represents an affine transformation, and we get its rotation angle with the `#radians` message.

```
Mobile>>heading
   ↑ location radians - Float halfPi
Mobile>>heading: aHeading
   self rotation: aHeading + Float halfPi
```

When a mobile is vertical, pointing to the top of the screen, its heading is -90° (-pi/2) in the screen coordinates system. In that situation the mobile is not rotated – or 0° rotated – therefore we add 90° (pi/2) to the heading to get the matching rotation angle of the mobile.

7.3.3 Torpedo

Alike a space ship, when a torpedo is just instantiated its nose points in the direction of the top of the screen and its vertices are given by the Figure 7.11.
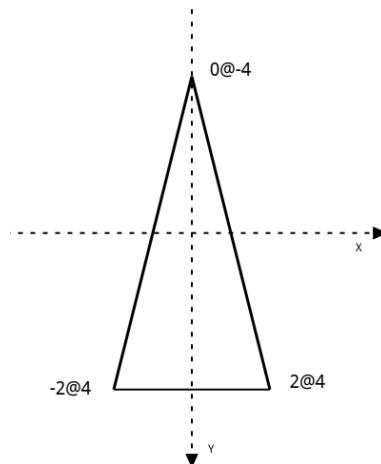
Figure 7.11: Torpedo diagram at game start-up

> *Given the vertices given by Figure 7.11, how will you write its* `morphExtent` *method?*

Exercise 7.4: Torpedo extent

> *How will you write the* `Torpedo`*'s* `drawOn:` *method?*

Exercise 7.5: Torpedo drawing

7.3.4 Drawing revisited

As you may have observed, the `SpaceShip` and `Torpedo` `drawOn:` methods share the same logic: drawing a polygon given its vertices. We likely want to push this common logic to their common ancestor, the `Mobile` class. It needs to know about its vertices, so we may want to add an instance variable `vertices` initialized in its subclasses with an array containing the points:

```
PlacedMorph subclass: #Mobile
    instanceVariableNames: 'acceleration color velocity vertices'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'

SpaceShip>>initialize
    super initialize.
    vertices ← {0@-15 . -10@15 . 0@10 . 10@15}.
    self resupply

Torpedo>>initialize
    super initialize.
    vertices ← {0@-4 . -2@4 . 2@4}.
    lifeSpan ← 500.
    acceleration ← 3000
```

However this is not a good idea. Imagine the game play with 200 torpedoes, the vertices array will be duplicated 200 times with the same data!

Class instance variable

In that kind of situation, what you want is a *class instance variable* defined in the class side – in contrast to the instance side where we have been coding until now.

We make use of the fact that all objects are instances of some class. The `Mobile class` is an instance of the class `Class`!

1. A *class instance variable* can be accessed and assigned only by the class itself in a *class method*.

2. An entity (i.e. a fired torpedo) can access class instance variables via *class methods*, by sending a message to a class (i.e. `Torpedo`) rather than to itself or some other entity.

3. In the class hierarchy, each subclass has its own instance of the class instance variable, and can assign a different value to it – in contrast with a *class variable* which is shared among all the subclasses (discussed later).

4. To edit the *class* instance variables and *class* methods, in the System Browser press the `class` button under the class list.

In the System Browser, we click the `class` button then we declare our variable in the `Mobile class` definition – Figure 7.12:

```
Mobile class
    instanceVariableNames: 'vertices'
```

Example 7.6: `vertices` an instance variable in `Mobile class`

Then we write an access method in the `Mobile class`, so `SpaceShip` and `Torpedo` instances can access it:

```
Mobile class>>vertices
    ↑ vertices
```
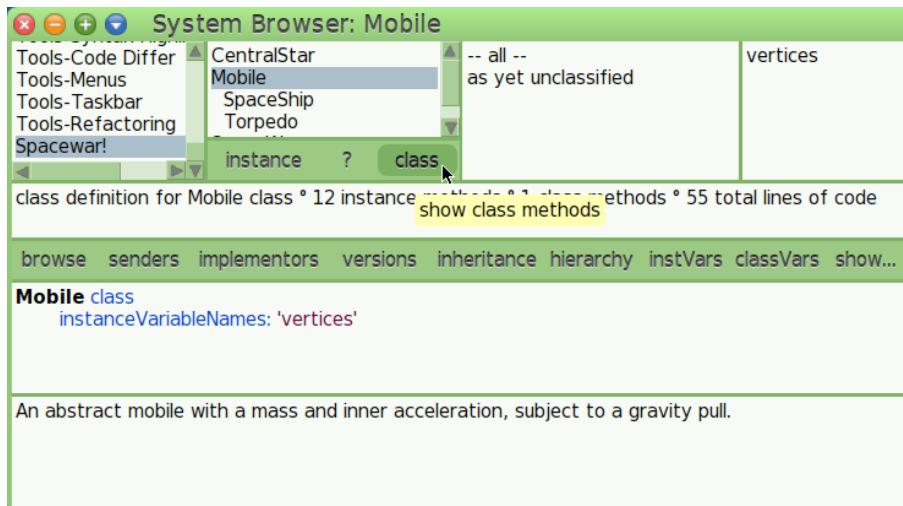

Figure 7.12: The class side of the System Browser

Next, each subclass is responsible to correctly initialize `vertices` with its `initialize` class method:

```
SpaceShip class>>initialize
"SpaceShip initialize"
    vertices ←  {0@-15 . -10@15 . 0@10 . 10@15}

Torpedo class>>initialize
"Torpedo initialize"
    vertices ← {0@-4 . -2@4 . 2@4}
```

Example 7.7: Initialize a class

When a class is installed in Cuis-Smalltalk, its `initialize` class method is executed. Alternatively select the comment and execute it with `Ctrl-d`.

Experiment in a Workspace to understand how a class instance variable behaves:

```
SpaceShip vertices.
⇒ nil
SpaceShip initialize.
SpaceShip vertices.
⇒ #(0@-15 -10@15 0@10 10@15)

Torpedo vertices.
⇒ nil
Torpedo initialize.
Torpedo vertices.
⇒ #(0@-4 -2@4 2@4)
```

Example 7.8: A class instance variable value is not shared by the subclasses

This is really the behavior we want: `SpaceShip` and `Torpedo` instances have a different diagram. However, every instances of a `SpaceShip` will have the same diagram, referring to the same `vertices` array (i.e. same location in the computer memory).

Each instance asks its class side with the `#class` message:

```
aTorpedo class
⇒ Torpedo
self class
⇒ SpaceShip
```

The `Torpedo`'s `drawOn:` is rewritten to access the vertices in its class side:

```
Torpedo>>drawOn: canvas
    | vertices |
    vertices ← self class vertices.
    canvas line: vertices first to: vertices second width: 2 color: color.
    canvas line: vertices third to: vertices second width: 2 color: color.
    canvas line: vertices first to: vertices third width: 2 color: color
```

How will you rewrite `SpaceShip`*'s* `drawOn:` *to use the vertices in its class side?*

Exercise 7.6: Space ship access to its diagram in class side

So far, we still have this redundancy in the `drawOn:` methods. What we want is `Mobile` to be responsible to draw the polygon given a vertices array: `self drawOn: canvas polygon: vertices`.

The `SpaceShip` and `Torpedo`'s `drawOn:` will then be simply written as:

```
Torpedo>>drawOn: canvas
    self drawOn: canvas polygon: self class vertices


SpaceShip>>drawOn: canvas
    | vertices |
    vertices ← self class vertices.
    self drawOn: canvas polygon: vertices.
    "Draw gas exhaust"
    acceleration ifNotZero: [
        canvas line: vertices third to: 0@35 width: 1 color: Color gray]
```

> *How will you write the* `drawOn:polygon:` *method in* `Mobile`*? Tip: use the iterator* `withIndexDo:`*.*

Exercise 7.7: Draw on `Mobile`

Class variable

A *class variable* is written capitalized in the argument of `classVariableNames:` keyword:

```
PlacedMorph subclass: #Mobile
    instanceVariableNames: 'acceleration color velocity'
    classVariableNames: 'Vertices'
    poolDictionaries: ''
    category: 'Spacewar!'
```

Example 7.9: `Vertices` a class variable in `Mobile`

As a class instance variable, it can be directly accessed from the class side and instances are grant access only with messages send to the class side. **Contrary** to a class instance variable, its value is common in the whole class hierarchy.

In Spacewar!, a class variable `Vertices` will have the same diagram for a space ship and a torpedo. This is not what we want.

```
SpaceShip>>vertices
    ↑ `{0@-15 . -10@15 . 0@10 . 10@15}`
```

7.3.5 Drawing simplified

Using a class variable in the present game design is a bit overkill. It was
an excuse to present the concept of class variables. If the game came with
an editor where the user redesigns the ship and torpedo diagrams, it would
make sense to hold the vertices in a variable. But our vertices of the space
ship and torpedo diagrams are constant. We do not modify them. As we
did with the mass of the space ship – Example 3.16 – we can use a method
returning a collection, surrounded with backtricks to improve efficiency.

```
SpaceShip>>vertices
    ↑ `{0@-15 . -10@15 . 0@10 . 10@15}`

Torpedo>>vertices
    ↑ `{0@-4 . -2@4 . 2@4}`
```

Example 7.10: Vertices returned by an instance method

Then in the drawing methods, we replace `self class vertices` by `self
vertices`.

7.3.6 Collisions revisited

In Example 4.23, we have a very naive approach for collision between the
central star and the ships, based on distance between morphs. It was very
inaccurate.

In bitmap games, a classic way to deal with collision detection is to look
at the intersections of the rectangles surrounding the graphic objects. The
`#displayBounds` message sent to a morph answers its bounds in the display,
a rectangle encompassing the morph given its rotation and scale.
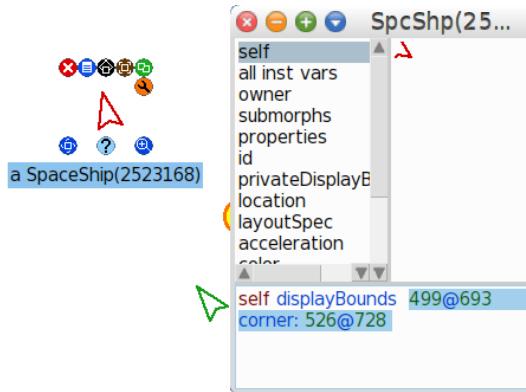
Figure 7.13: The display bounds of a space ship

When browsing the `Rectangle` class, you learn the `#intersects:` message tells us if two rectangles overlap. This is what we need for a more accurate collision detection between the central star and the space ships:

```
SpaceWar>>collisionsShipsStar
    ships do: [:aShip |
       (aShip displayBounds intersects: centralStar displayBounds) ifTrue: [
          aShip flashWith: Color red.
          self teleport: aShip]]
```

Example 7.11: Collision (rectangle overlapping) between the ships and the Sun

Nevertheless we use the VectorGraphics framework, its knows about the shape of each morph it is rendering. It uses this knowledge for collision detection at the pixel precision. We rewrite our detection method simpler with the `#collides:` message:

```
SpaceWar>>collisionsShipsStar
    ships do: [:aShip |
       (aShip collides: centralStar) ifTrue: [
          aShip flashWith: Color red.
          self teleport: aShip]]
```

Example 7.12: Collision (pixel precision) between the ships and the Sun

Rewrite the three collision detection methods between space ships, torpedoes and the central star.

Exercise 7.8: Accurate collision detection

8 Events

> When I used to read fairy tales, I fancied that kind of thing never happened, and now here I am in the middle of one!
> —*Lewis Carroll, Alice in Wonderland*

What just happened?

We talked above about control flow, how one makes decisions about what to do in making calculations. We talked about this like the entire computer processing resource was dedicated to this task. But it isn't so.

Computers may be fast at some calculations, but they are only so fast, and when there are many things to do, one shares and takes turns.

So aside from *do this and then do that*, *event*s happen.

Also, a computer may be fast enough that it literally has *nothing* to do. What does it do then? When a processor *goes to sleep*, how do we get its attention?

You are reading the right chapter to know.

8.1 System Events

Modern integrated *System On a Chip* (*SOC*) hardware has many circuits which are active at the same time. So one kind of event is sensing something happening in the world. Class `EventSensor` handles keyboard key press and mouse *hardware interrupts*, translating between hardware signals and software event objects.

Basically, a morph *raises its hand* and says what events, if any, it is interested in receiving. Then it implements methods to get the event objects holding the information of the captured events. In Cuis-Smalltalk, the class `MorphicEvent` and its subclasses represent the diversity of events in the system.

```
MorphicEvent
  DropEvent
  DropFilesEvent
  UserInputEvent
    KeyboardEvent
    MouseEvent
      MouseButtonEvent
```

```
   MouseMoveEvent
   MouseScrollEvent
 WindowEvent
```

As `MouseMoveEvent`s are generated, the `HandMorph` adjusts its screen position. When mouse and keystroke events arrive, the `HandMorph` coordinates the "dispatch" of events to the proper morph under the hand as well as displaying tool tips and carrying morphs in transit during drag operations.

As we saw in the previous chapter with `ColorClickEllipse`, any morph may override default `Morph` methods to assert that it handles various user events and the methods which take the associated event objects when events arrive.

Basically, user input events are generated, a `HandMorph` reflects any cursor movement, morphs react to events, each long running task gets a time slice and makes some progress, any display changes are updated on the screen, and the next step happens. Time marches forward a step.

This happens over and over and over, keeping the juggler's illusion that all balls in the air are moving at once. Underneath, the balls are each moving just a bit, in sequence.

8.2 Overall Mechanism

In Section 6.4 [A brief introduction to Inspectors], page 94, we explained how to set properties for an individual morph instance to handle a specific event. In this case, one property informed Cuis-Smalltalk we were interested in a given event (`#handlesMouseDown`), a second property defined the behavior with a block of code to be executed each time this event occurred.

Alternatively, to handle events in a given morph class, we define the behavior with instance methods. In the `Morph` class, observe the method categories `event` and `event handling testing`.

Method catrgory `event handling testing` lists methods returning a Boolean to indicate if the instance should be notified about certain events. Let's take a look at one of these methods, `handlesMouseDown:`, its comment is worth reading:

```
Morph>>handlesMouseDown: aMouseButtonEvent
"Do I want to receive mouseButton messages ?
- #mouseButton1Down:localPosition:
- #mouseButton1Up:localPosition:
- #mouseButton2Down:localPosition:
- #mouseButton2Up:localPosition:
- #mouseButton3Down:localPosition:
- #mouseButton3Up:localPosition:
- #mouseMove:localPosition:
- #mouseButton2Activity
NOTE: The default response is false. Subclasses that implement these
```

```
messages directly should override this one to return true.
Implementors could query the argument, and only answer true for (for
example) button 2 up only."
"Use a property test to allow individual instances to dynamically
specify this."

    ↑ self hasProperty: #'handlesMouseDown:'
```

As defined by default, this method and the other handlers check to see if an instance has defined a property with the same name as the standard method. So each individual instance can add its own behavior.

In a morph class where we want *all* instances to handle mouse down events, we just override the appropriate method to return true:

```
MyMorph>>>>handlesMouseDown: aMouseButtonEvent
    ↑ true
```

Now in the `events` method category for class `Morph`, we find the handlers listed in the comment above. A `ScrollBar`, a kind of Morph to represent a list's position control, scrolls its list contents when a mouse button 1 is pressed:

```
ScrollBar>>mouseButton1Down: aMouseBtnEvent position: eventPosition
"Update visual feedback"
    self setNextDirectionFromEvent: aMouseBtnEvent.
    self scrollByPage
```

To discover other events available for your morph, explore with the System Browser as described above.

8.3 Spacewar! Events

Obviously our Spacewar! game handles events. Firstly we want to control the ships with the keyboard. Secondly, we want the game to pause when the mouse cursor moves out of the game play, and resume when it enters again.

In our design, a unique morph, `SpaceWar` instance, models the game play. Therefore we want this instance to handle the events described above.

8.3.1 Mouse event

Mouse cursor enters game play

We want to catch events when the mouse cursor moves over our `SpaceWar` morph.

> Which method should return true to let the game play be notified with a dedicated messages that the mouse cursor enters or leaves? In which class should we implement this method?

Exercise 8.1: Get notified of mouse move-over event

Once we make explicit we want the game play to receive mouse move-over events, we need to set the behavior accordingly with dedicated methods.

Each time the mouse cursor enters the game play, we want to:

- **Get keyboard focus.** It follows the mouse cursor: the keyboard input goes to the morph under the mouse cursor. In Cuis-Smalltalk, the mouse cursor is modeled as a `HandMorph` instance, an event object (see event classes hierarchy at the beginning of this chapter). An event object is interrogated about its hand with the `#hand` message. All in all, we want the keyboard focus to be targeted toward our game play when the mouse enters:

  ```
  event hand newKeyboardFocus: self
  ```

- **Resume the game.** The continuous update of the game is done through a dedicated process stepping mechanism, which will be discussed in the next chapter. The game play just asks itself to resume stepping:

  ```
  self startStepping
  ```

> Which message is sent to the game play to be notified the mouse cursor enters the game play area? How should the matching method be written?

Exercise 8.2: Handle mouse enter event

Mouse cursor leaves game play

We also want to be informed when the mouse cursor leaves our `SpaceWar` morph. Thanks to the work done in Exercise 8.1, we already informed Cuis-

Smalltalk we want to be notified of mouse movement over the game play. However we need to code the behavior when the mouse cursor leaves the game play:

- **Release keyboard focus.** We tell Cuis-Smalltalk the game play does not want keyboard focus:

    ```
    event hand releaseKeyboardFocus: self
    ```

- **Pause the game.** We stop the continuous *stepping* update of the game:

    ```
    self stopStepping
    ```

Which message is sent to the game play to be notified the mouse cursor leaves the game play area? How should we write the overridden method?

Exercise 8.3: Handle mouse leave event

In graphic user interface, a visual effect is often used to inform the user the keyboard focus changed. In Spacewar! we change the game play background depending on the sate of the keyboard focus.

In Figure 8.1, at the left keyboard focus is on the game; at the right keyboard focus is not on the game, it is paused and we can see underneath.
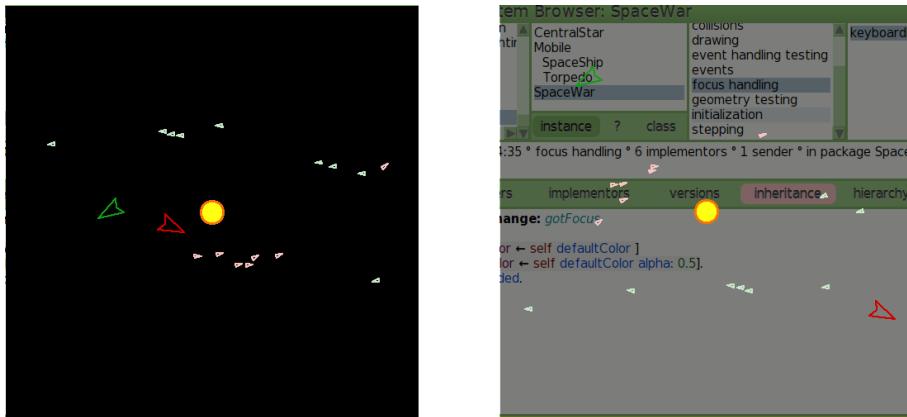
Figure 8.1: Spacewar! effect depending on the keyboard focus

In the Morph framework, the `#keyboardFocusChange:` message is sent to the morph losing or gaining the keyboard focus, its parameter is a Boolean. Therefore we implement the Figure 8.1 behavior in the matching `SpaceWar`'s method `keyboardFocusChange:`

```
SpaceWar>>keyboardFocusChange: gotFocus
   gotFocus
      ifTrue: [color ← self defaultColor]
      ifFalse: [color ← self defaultColor alpha: 0.5].
   self redrawNeeded
```

Example 8.1: Spacewar! keyboard focus effect

8.3.2 Keyboard event

To control the space ships, we use the keyboard. Therefore we want the game play to be notified of keyboard events.

> *Find out which method should return true to let the game be notified of keyboard event.*

Exercise 8.4: Get notified of keyboard event

We can decide to be notified of the key down or key up event and also key down then up event (*key stroke*). As long as our `SpaceWar` morph responds true to the `#handlesKeyboard` message, it receives the messages `#keyUp:`, `#keyDown:` and `#keyStroke:`. By default, the matching methods in the `Morph` class do nothing.

The argument of these messages is a `KeyboardEvent` object to which, among other things, you can ask the `#keyCharacter` of the pressed key or test about some special keys as the keyboard arrows. The first player ship – the green one – is controlled with the keyboard arrows when there are stroked:

```
SpaceWar>>keyStroke: event
  event isArrowUp ifTrue: [↑ ships first push].
  event isArrowRight ifTrue: [↑ ships first right].
  event isArrowLeft ifTrue: [↑ ships first left].
  event isArrowDown ifTrue: [↑ ships first fireTorpedo].
  ...
```

Example 8.2: Keystroke to control the first player ship

To control the second player ship, we use another classic arrangement in QWERTY keyboard controlled game: WASD[1].

Append the additional code to Example 8.2 to control the second player ship with the keys WASD. As a reminder, in Smalltalk the character code for q can be written as `$q`.

Exercise 8.5: Keys to control the second player ship

---

[1] https://en.wikipedia.org/wiki/Arrow_keys#WASD_keys

9 Code Management

> Change is easy, except for the changed part.
> —*Alan Kay*

Regarding the source code, Cuis-Smalltalk comes with several tools to manipulate it: the image, the change record, the change set and the package system. We give you a tour around these mechanisms then explain how you should manage the code of an application written with Cuis-Smalltalk.

9.1 The Image

We already wrote about the Cuis-Smalltalk *image* (See Section 1.2 [Installing and configuring Cuis-Smalltalk], page 7). When saving the state of the virtual machine in the image file, every single change done in the environment will be embodied in the saved image: this includes the windows in the environment, Workspace contents, newly written classes and methods, existing instances including the visual morphs, a debugging session with a System Browser, an Inspector, etc.

At any time, the user can save the image with ...World menu → `Save...` Alternatively `Save as...` saves the image under an alternate name provided by the user.

Saving the image is the easiest and most straightforward method to save your own code. But we can't really call that code management as your code is not saved in a dedicated file of its own but mixed into other code in an image. Moreover it will be unpractical to share your work with others, for example via a version control system.

For various reasons, an image may be in fuzzy state: the virtual machine may crash when running it, the file system may be unstable, or the environment may be in a lock down state. This is a drawback when using the image as your sole source code repository. The net result could be loosing your work.

Where you lost code because of a virtual machine crash there is a solution to recover your lost editing, the *Change Log*.

9.2 The Change Log

Cuis-Smalltalk records any action occurring in the environment: the code you edit in the System Browser, the code you execute in a Workspace. There-

fore, in the event of a Cuis-Smalltalk crash you can restore unsaved changes when you launch the same Cuis-Smalltalk image again. Let's explore this feature with a simple example.

On a fresh Cuis-Smalltalk installation, create a new class category named `TheCuisBook` and within `TheBook` class:

- Over the class category pane of System Browser (at the most left), do ...Right click → `add items...` (a)... key in `TheCuisBook`.

- Select this new class category and create the class `TheBook` as a kind of `Object`: select the `TheCuisBook` category then in the source code below edit the class template to replace `#NameOfClass` with `#TheBook` then save the class definition with *Ctrl-s*.

Open a Workspace, then key in the following code:

```
| myBook |
myBook ← TheBook new
```

Cuis-Smalltalk does not save code you key in the Workspace, but code you execute. Let's execute this code: *Ctrl-a* then *Ctrl-p*, the Workspace prints the result: `a TheBook`, an instance of a `TheBook` class.

Now kill Cuis-Smalltalk abruptly. On GNU/Linux, you can use the `xkill` command to terminate Cuis-Smalltalk by clicking at its window.

Now start Cuis-Smalltalk again, and it immediately informs you there are unsaved changes:
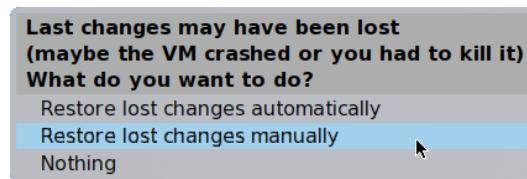
Figure 9.1: Cuis-Smalltalk informs about lost changes

From there you have three options:

- **Restore lost changes automatically.** Cuis-Smalltalk will apply all the changes: new class definitions, new methods; edited class definitions and method source code; executed code (in Workspaces or any places where code can be executed). Often this is not really what you want to do, particularly the executed code.

- **Restore lost changes manually.** In the subsequent `Lost changes` window

you are presented with the unsaved changes, one per line, in chronological order, with the older ones at the top of the list. You select each change (line) you want to restore, then you apply your selection with the `file in selections` button.
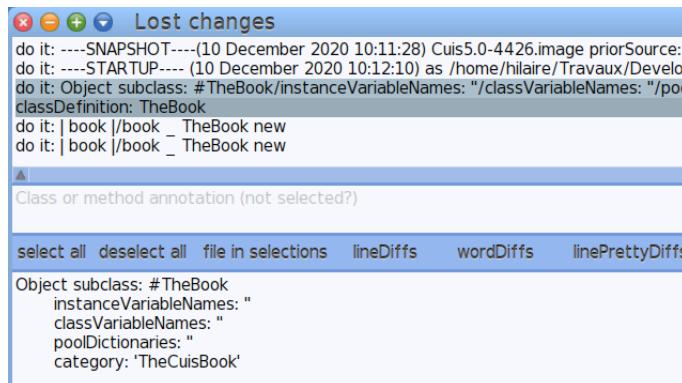
Figure 9.2: Manually select the changes to file in

To file-in the changes related to the creation of the `TheBook` class but not the executed code in the Workspace, select the two lines related to class definition.

The contextual menu (mouse right click) of the `Lost changes` window offers a lot of options to filter the changes. Useful when the batch of lost changes is important.

- **Nothing.** No changes are restored. Keep in mind that unsaved changes aren't discarded until you save your image.

    In case you change your mind and you want to recover changes, do ...World menu → `Changes...` → `Recently logged Changes...`.

    The system presents you a list of image snapshots tagged with a date stamp. Pick up the one occurring just before you lost your code, most likely at the top of the list. Then in the `Recent changes` window, you proceed as described earlier to cherry pick the changes to restore.

9.3 The Change Set

On a fresh Cuis-Smalltalk installation, each code you edit in the System Browser is recorded in a *Change Set*,

You browse a change set with a tool named the *Change Sorter*: ...World menu → `Changes...` → `Change Sorter`...
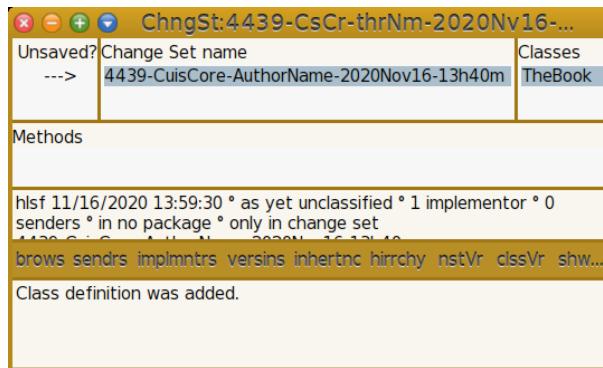
Figure 9.3: The Change Sorter, class edit

The `TheBook` class we added to Cuis-Smalltalk in the previous section is a change made to the core of the system. By default, it is recorded in a change set automatically created by the system. In Figure 9.3 at the top right, observe the class `TheBook`, it belongs to a change set named `4439-CuisCore-AuthorName-2020Nov16-13h40m`. In the left pane, each unsaved change set is marked with a `--->`. Here it tells us the change was not saved on disk. To save the change set, just use its contextual menu and use one of the `file out` entries. The change set will be saved along the Cuis-Smalltalk image under its system name with `AuthorName` substituated with the real author name.
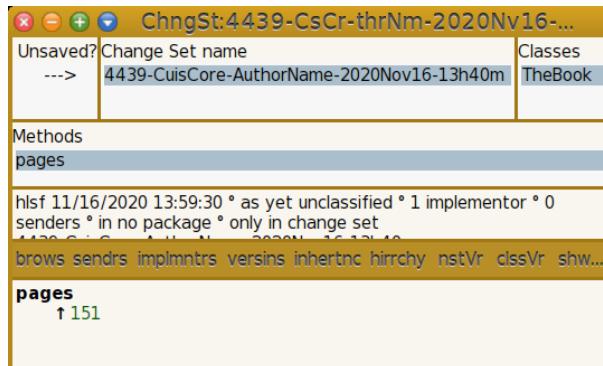
Figure 9.4: The Change Sorter, method edit

Observe Figure 9.4, after we added the method `pages` to the `TheBook` class, the middle pane lists the added or modified methods. When a method is selected its source code is printed in the bottom pane.

Let's say we save the change set – `File out` entries in the change sorter tool menu. This creates a new file `4451-CuisCore-HilaireFernandes-2020Nov14-21h08m-hlsf.001.cs.st` along the Cuis-Smalltalk image file:

```
From Cuis 5.0 [latest update: #4450] on 18 November 2020 at 9:05:09 am'!
!classDefinition: #TheBook category: 'TheCuisBook'!
Object subclass: #TheBook
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'TheCuisBook'!

!TheBook methodsFor: 'as yet unclassified' stamp: 'hlsf 11/18/2020 09:04:58'!
pages
    ↑ 151! !
```

Example 9.1: Change set contents

To load this change set back in a new image, you use the *File List* tool ...World menu → `Open` → `File List`... Browse the folder containing the change set file to load, then select it, from there you have three options to manipulate it.
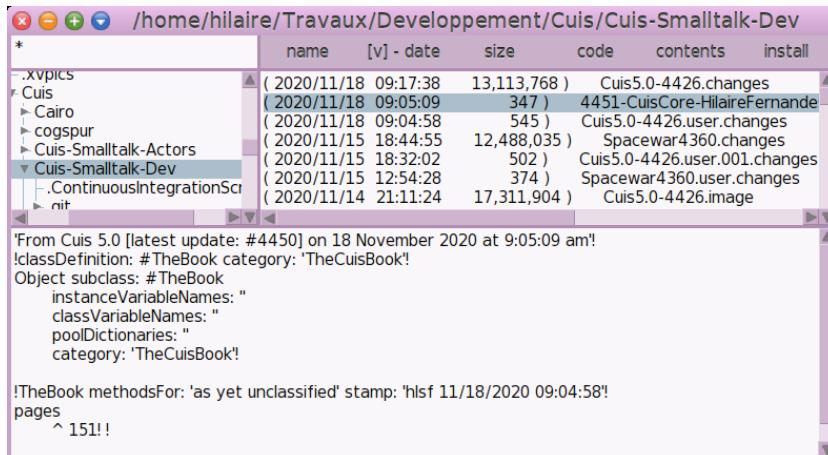
Figure 9.5: The File List tool, to install a change set and more

- **code**. It opens a kind of System Browser limited to the code in the change set file. It is a very handy tool to read and to learn the code from the change set.

- **contents**. It opens a *Change List* tool to review the modifications to the image this change set will produce once installed. It also let you cherry pick the individual changes you want to install and to discard. Each line you cherry pick represents a class or a method addition/modification. Once you select the code to install, press the `file in selections` button to proceed with the installation.

  Consider a co-developer modifying the `TheBook` class, she added an instance variable `pages` and adjusted the `pages` methods accordingly. She filed out her changes then shared the file with you. Observe in Figure 9.6 how you will review her changes with the Change List tool. Our code that will be removed from the image is shown in red stroked, and her code that will be installed is shown in green.

Figure 9.6: Change List tool to review modifications to the image

- **install**. It just installs the complete change set without interactivity.

The change set way of managing the source code is used by the developers of Cuis-Smalltalk, to work on its core image. When you want to write an application, a dedicated tool or even a set of classes covering a specific domain, you really want to use something else to manage the code: a package.

9.4 The Package

A package can hold a set of classes part of the same class category.

Let's save our `Morphic-Learning` category as a package using our Installed Packages Browser.

...World menu → `Open...` → `Installed Packages`...
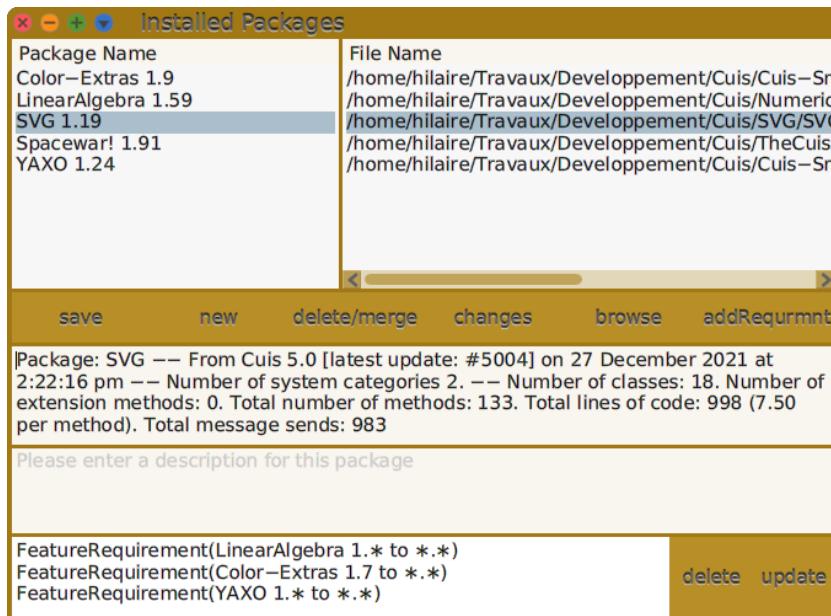
Figure 9.7: Installed Packages Browser

Note that we invoked `Feature require: 'SVG'` earlier. Looking at the package names, we can observe several things of note:

- Each package is versioned; `SVG` has version 1, revision 19.

- There are packages `YAXO`, `Color-Extras` and `LinearAlgebra` which we never asked for.

- Looking at the lower pane, note that those packages are **required** by the package `SVG`.

Now this is important. When a packaged Feature is required, it may specify that it also requires other packages to work properly, and in fact to specify that those packages be up to a specific version level.

This is the key to being able to safely compose packages which have requirements.

OK, let's click on the `New` button and type `Morphic-Learning` into the prompt. This results in a new package with the same name as our `Morphic-Learning` category. Note that this is version 1, revision 0 (1.0 at right) and that the package has yet to be saved.
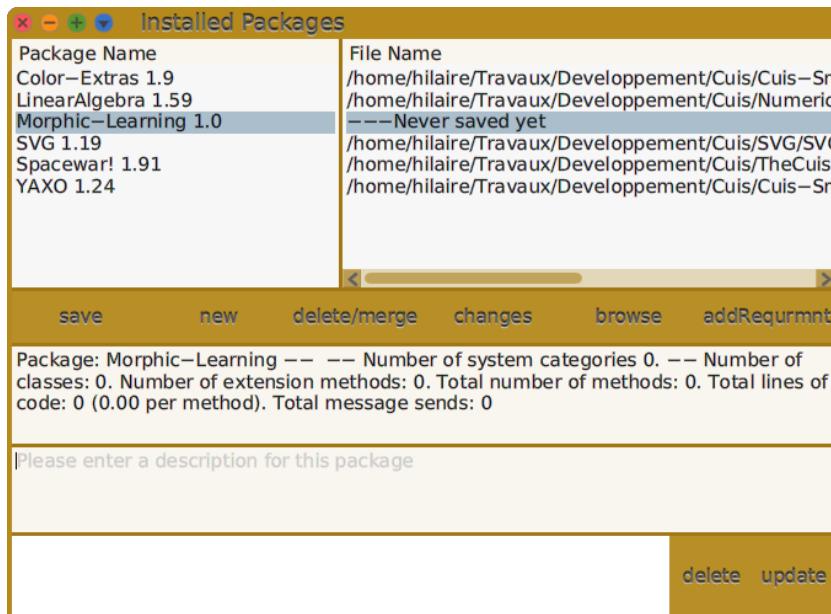
Figure 9.8: New Package – `Morphic-Learning`

Let's say that in our clock morph we want to use color named as in the `Color-Extra` package. So to be able to load our `Morphic-Learning` package which makes use of this we need to select our new package and click on the `add Requirement` button at center, right.

This brings up a list of loaded packages to choose from.
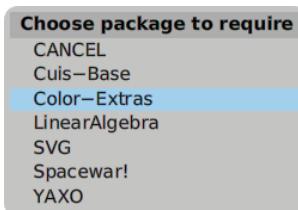
Figure 9.9: Select package (or Cuis base version) to require

Now when we `save` our package, we see the pathname where the package file was created. We can now safely email this file, check it into a version control system, make a copy to our backup thumb drive.
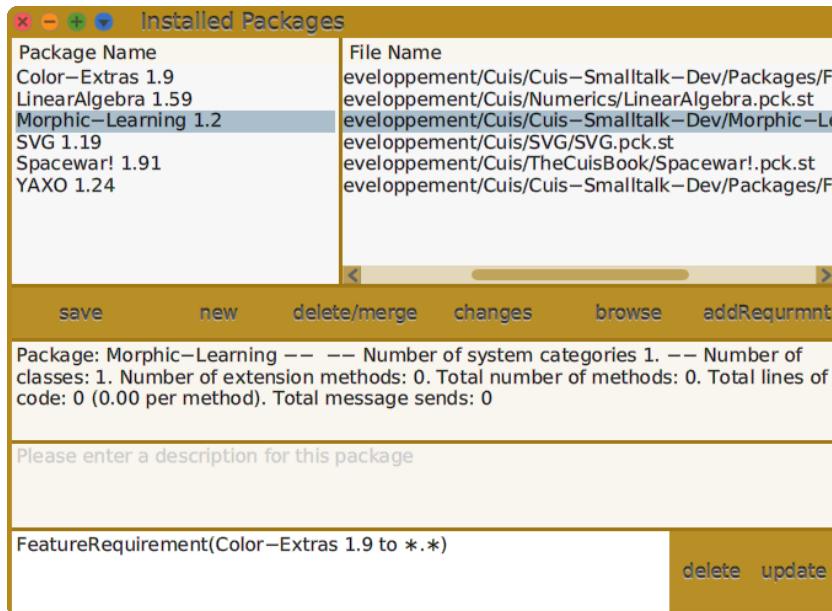
Figure 9.10: Saved package – `Morphic-Learning`

As mentioned above, package files are just text files with a special format which Cuis-Smalltalk knows how to load. If you open a File List browser and view the package file, you will see information on how the package was created, what it provides and requires, and, if you filled in the comment box in the Installed Packages browser, a description.

> *Type "Morphic Toys" into the comment box, re-save your package, and (re)select the package in a File List to see your package description.*

Exercise 9.1: Describe a package

> *If you have not already done so, create and save a [Spacewar! package], page 28. There are no additional requirements to specify.*

Exercise 9.2: Save Spacewar! package

There are other interesting things we can do with packages. We can include several class categories in a single package. Consider we want to span our CuisBook classes in two categories `TheCuisBook-Models` and `TheCuisBook-Views`. A new package created with the name `TheCuisBook` includes these two class categories; this label is a *prefix* to search for the matching categories to include in the package.

Therefore, often, a package comes with several categories to organize the classes in matching domains. We encourage to do so. When an application or framework grows, to keep a sound organisation, you may fell the need to reshape the class categories: rename, split, merge, etc. As long as you keep the same prefix in the class categories and the package name, your classes will be safe in the same package. In the System Browser, you can drag and drop any class in any class category to reorganize.

> *Create a `TheCuisBook` package from the two class categories `TheCuisBook-Models` and `TheCuisBook-Views`. The former contains a `TheBook` class and the later a `TheBookMorph` class. Save the package on disk.*

Exercise 9.3: Two class categories, one package

Imagine we need to print the page number of the `TheBook` table of contents as lower cased roman number, as we do with the printed version of this book. The code is very simple:

```
4 printStringRoman asLowercase
⇒ 'iv'
```

Instead of invoking this sequence of messages each time we need it, we add a dedicated message to the `Integer` class:

```
Integer>>printStringToc
    ↑ self printStringRoman asLowercase
```

Now within our `TheBook`'s methods we just do things like:

```
../..
 aPage ← Page new.
 aPage number: 1 printStringToc.
 ../..
```

Now we are facing a problem. For the need of the `TheBook` package
we extend the `Integer` class with a method `printStringToc`, however this
method addition is part of the Cuis-Smalltalk core system and its associated
default change set. See Figure 9.11, the Change Sorter tool exactly shows
that.
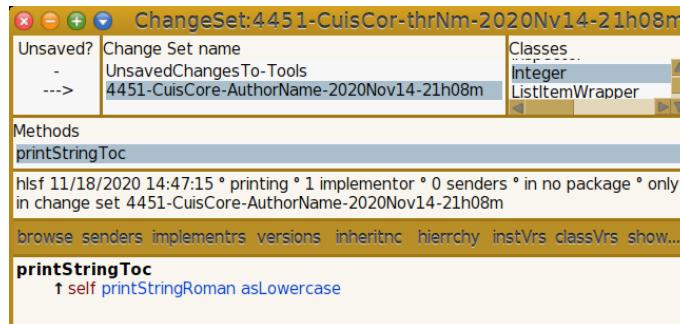
Figure 9.11: Change Sorter, supplementary method to core

Therefore when saving our `TheBook` package this method is not included
and it is lost when quitting Cuis-Smalltalk. To include it in our pack-
age we categorize it in a method category with the `*TheCuisBook` pre-
fix. `*TheCuisBook-printing` is a good candidate. In the System Browser
method pane, over `printStringToc`, do ...Contextual menu → `more...` →
`change category...` and key in `*TheCuisBook-printing`.

Now the Change Sorter writes about `Integer>>printStringToc`:
Method was moved to some other package. The Installed Packages tools
now tells us we have an extension, use its `browse` button to get an update
on the package contents.
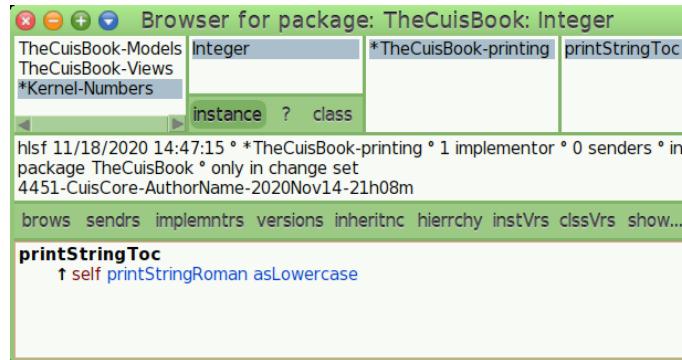
Figure 9.12: Package with extension to the `Integer` class of the `Kernel-Numbers` system class category

Observe how each category – class or method one – of an extension is prefixed with a `*`.

9.5 Daily Workflow

For our Spacewar! game, we created a dedicated package `Spacewar!.pck.st` file. This is the way to go when writing external package: define a dedicated package and from time to time save your work with the `save` button in the Installed Packages tool (See Figure 2.3).

Cuis-Smalltalk uses GitHub to host, version, diff its core development and to manage a set of external packages (i.e. code that is maintained independently and outside Cuis-Smalltalk but closely related to it).

Package files are simple text files, encoded for latin alphabet (ISO 8859-15) and handled without problems by GitHub. Cuis-Smalltalk uses the LF (ascii code 10) newline convention, as preferred in GitHub. This allows Git/GitHub to diff versions, and merge branches.

Separate GitHub repositories are used for projects, i.e. package or set of closely related packages that are always loaded and maintained together as a whole.

Your daily workflow with Cuis-Smalltalk to develop an external package will look like:

1. Start with a standard, fresh, Cuis image. Never save the image.
2. Set up your preferred version control system to manage your external packages. The recommendation is to use a GitHub repository with a name beginning with 'Cuis-Smalltalk-', so it will be easy for anybody

to find it. But beside this consideration, using any other version control system is fine.

3. Install the necessary packages from the Cuis-Smalltalk Git repositories.

4. Develop. Modify and/or create packages.

5. Save own packages (to your preferred repositories).

6. add / commit / push accordingly to your version control system

7. Fileout changes that are not part of any package. These are automatically captured in numbered changesets, separated from changes to packages.

8. Exit the image. Usually without saving.

> In addition to adding a package preload *requirement*, you can also select a requirement and `delete` or `update` it using the buttons at the lower right. Sometimes a package changes which your code depends on and you have to change your code to accord. When this happens, to want to be sure to require the newer, changed version. Selecting a requirement and pressing `update` will update the requirement to use the latest loaded package version.

9.5.1 Automate your image

As described in the daily workflow, it is a good habit to not save the whole image but only the modified package of the edited source code. However, each time we start a coding session, it is tedious to set up the image to fit our personal needs and taste.

Things one may want to personalize in the image are:

- Preferences adjustments,
- Placement of tools like System Browser, Workspace, Transcript,
- Default contents in the Workspace, ready to be executed,
- Installation of Packages.

We want to record these image preferences in a `setUpEnvironment.st` script to be executed at start up. On GNU/Linux, you ask Cuis-Smalltalk to run a script with the `-s`, for example `squeakVM Cuis5.0.image -s setUpEnvironement.st` where `setUpEnvironement.st`. is a file containing Smalltalk code. A real life example may look like:

```
../cogspur/squeak Cuis5.0-4426 -s ../scripts/setUpEnvironment.st
```

We describe in detail an example of a set up script organizing the environment as seen in Figure 9.13. It is interesting Smalltalk code poking

around heterogeneous areas of Cuis-Smalltalk like the developer tools, the Morph system, the preferences and collection handling.
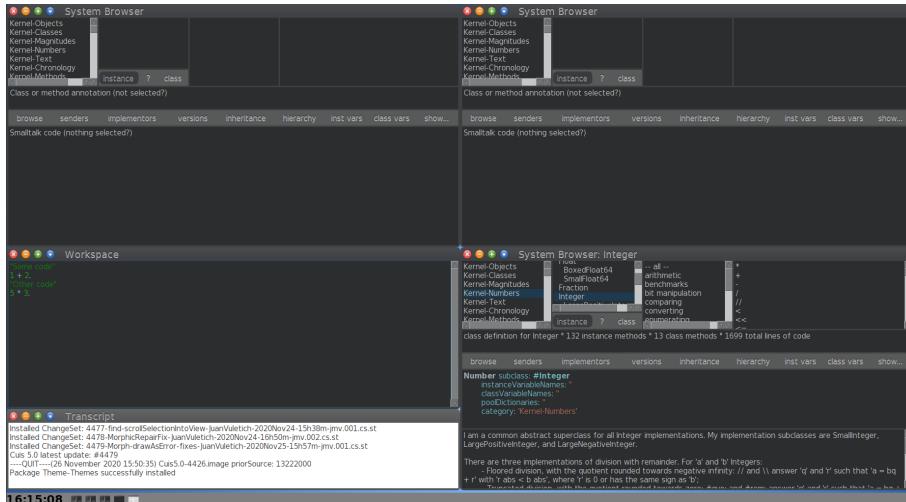
Figure 9.13: Environment of an image started with the set up script

Let's start by removing the open windows:

```
| list |
"Delete all windows but the taskbar"
self runningWorld doOneCycleNow.
list ← UISupervisor ui submorphs reject: [:aMorph |
    aMorph is: #TaskbarMorph].
list do: [:each | each delete].
```

The whole user interface world of Cuis-Smalltalk is a kind of Morph, a `WorldMorph` instance. Its submorphs are windows, menus, the taskbar or any kind of morph the user can interact with. To access this `WorldMorph` instance you ask to the `UISupervisor` with the `#ui` message. Once we select all the morphs in the world but the taskbar – really `#reject:` it – we `#delete` them from the world.

Next, we change the preferences:

```
| list morph |
../..
"Change to Dark theme"
Feature require: #'Theme-Themes'.
DarkTheme beCurrent.
```

```
"Adjust font size"
Preferences at: #defaultFontSize put: 12.
"Adjust taskbar size"
morph ← UISupervisor ui taskbar.
morph scale: 1 / 2.
Display fullScreenMode: true.
self runningWorld doOneCycleNow.
```

We require Theme-Themes package; as it is not installed on the default image, it will be searched on the disk for installation. Regarding the taskbar access, remember we deleted all the morphs but the taskbar from the world, therefore the taskbar is really the first in the sub morphs collection of the world. Finally, we update the scale of the taskbar and we request Cuis-Smalltalk to display itself full screen. As we changed several visual parts, we request one update cycle of the environment (i.e. the running world in Cuis-Smalltalk terminology).

Before installing the tools, we ask a `RealEstateAgent` the free area. Sadly this agent does not take into consideration the area occupied by the taskbar, so we need to tweak its answer. Then we compute a quarter of this free area extent (half in width and half in height make a quarter of the whole free area):

```
| list morph area extent |
../..
"Compute the available free space for windows placement"
area ← RealEstateAgent maximumUsableArea
   extendBy: 0 @ morph morphHeight negated.
extent ← area extent // 2.
```

Now we are ready to install a few tools. First three browsers each occupying a quarter of the screen:

```
"Open a few System Browsers"
BrowserWindow openBrowser
   morphPosition: 0 @ 0;
   morphExtent: extent.
BrowserWindow openBrowser
   morphPosition: area width // 2 @ 0;
   morphExtent: extent.
"Open a System Browser on a specific class"
morph ← BrowserWindow openBrowser
   morphPosition: area extent // 2;
   morphExtent: extent.
morph model setClass: Integer selector: nil.
```

Then in the remaining free quarter, we install a workspace occupying two thirds of the area and a transcript one third. The workspace is installed with

some default contents. We need to hack a bit because when asking for a new Workspace, Cuis-Smalltalk does not answer the created instance, we have to search it in the windows of the world.

```
"Open a Workspace with some default contents"
morph ← Workspace openWorkspace.
morph model actualContents: '"Some code"
1 + 2.
"Other code"
5 * 3.'.
morph
   morphPosition: 0 @ (area height // 2);
   morphExtent: extent x @ (2 / 3 * extent y).
"Open a transcript for logs"
TranscriptWindow openTranscript
   morphPosition: 0 @ (area height // 2 + (2 / 3 * extent y));
   morphExtent: extent x @ (1 / 3 * extent y).
```

Of course you should adjust the argument of the `#actualContents:` message to meaningful code for your usage.

9.6 Source Code File

In this chapter, you encountered files produced by the Cuis-Smalltalk management tools. Some come with the extension `.pck.st` and others with the extension `.st`. They both contain Smalltalk code but serve different purposes.

Files with extension `.pck.st` are Cuis-Smalltalk package files. They have a preamble with package information, like feature provided, version information, a description for the package and the package requirements.

Smalltalk code files can be installed by selecting them in the File List Browser and clicking on `install` button.

When a package is installed, its dependencies are also installed and the loaded packaged appears in the Package List tool. Package files `.pck.st` are created using the Packages List tool in Cuis-Smalltalk.

Files with extension `.st` contained serialized Smalltalk code – classes and methods. They are created when you *file out* a Smalltalk system category, class or method. You can serialize code in the System Browser by selecting a **system category**, **class** or **method** → right click and select `fileOut` from the context menu.

Unlike `.pck.st` files, `.st` files don't contain any dependencies, description, or package information, only serialized Smalltalk code. `.st` files are around since Smalltalk's early days, while `.pkg.st` were added with the Cuis-Smalltalk package functionality.

In addition to `install`, selecting a code file in the File List Browser also adds buttons to inspect the `code` and to treat code `contents` like a Change Sorter. When inspecting the code, a browser shows what code is different between the file and the running image and allows one to import individual classes or methods with the help of the context menu.

10 Debug and Exception Handling

Reactive Principle: Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.
    —*Dan Ingalls*

The quote above is worth repeating.

We think of Morphs and "data objects" as able to present themselves to be inspected, but Smalltalk's runtime state is also presentable.

10.1 Inspecting the Unexpected

We have seen how various exceptional situations cause the appearance of a debugger window. Indeed, `Exceptions` are also objects which remember their context and can present it. Above, we have seen how to generate `MessageNotUnderstood` and `ZeroDivide` Exception instances.

This is another area where the actual mechanics are complex, but the basic ideas are simple.

Exception instances, being objects, also have classes. The `BlockClosure` has a method category `exceptions` which gathers some handy methods which allow one to `ensure:` cleanup or capture and use exceptions (`on:do:` and friends).

```
FileEntry>>readStreamDo: blockWithArg
  "Raise FileDoesNotExistException if not found."
  | stream result |
  stream ← self readStream.
  [ result ← blockWithArg value: stream ]
ensure: [ stream ifNotNil: [ :s | s close ]].
  ↑ result
```

Example 10.1: Ensure a FileStream is closed

Exceptions are created and *signal*ed. Let's make one and look at it.
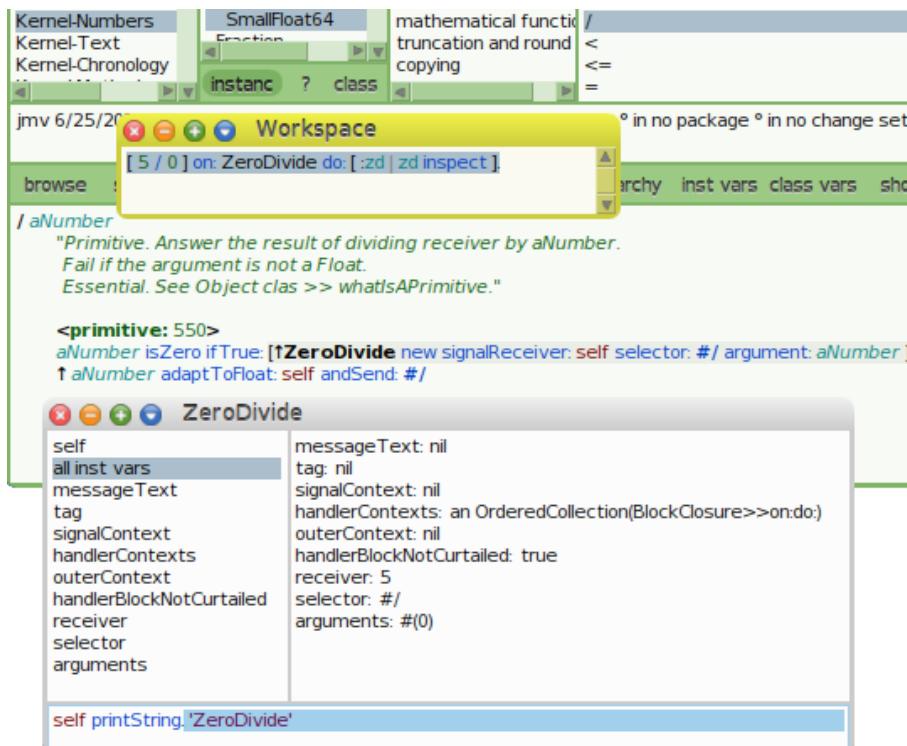
Figure 10.1: Inspecting a `ZeroDivide` instance

Again, we can use an Inspector on any object, and `Exception`s are no exception! Now you know how to capture one when you need to.

`Exception`s, like `MorphicEvent`s are a change, an exception, to typical control flow.

We noted above the special pseudo-variable, `thisContext`. Signalling an exception captures this.

```
Exception>>signal

    ↑ self signalIn: thisContext
```

Example 10.2: Capture thisContext

Just as Smalltalk code has special view windows which we call `Browsers`, `Exceptions` have an enhanced viewer we call the `Debugger`. Let us look at how to use this very useful viewer.

10.2 The Debugger

First, we need a fairly simple code example to look at. Please type or copy the following into a Workspace.

```
| fileNames |
fileNames ← OrderedCollection new.
(DirectoryEntry smalltalkImageDirectory)
    childrenDo: [ :f | fileNames add: f name ].
fileNames asArray.
```

Example 10.3: Names of Directory Entries

Now, you can `Ctrl-a` (*select* **A***ll*) and `Ctrl-p` (*select* **P***rint-it*) to see the result.
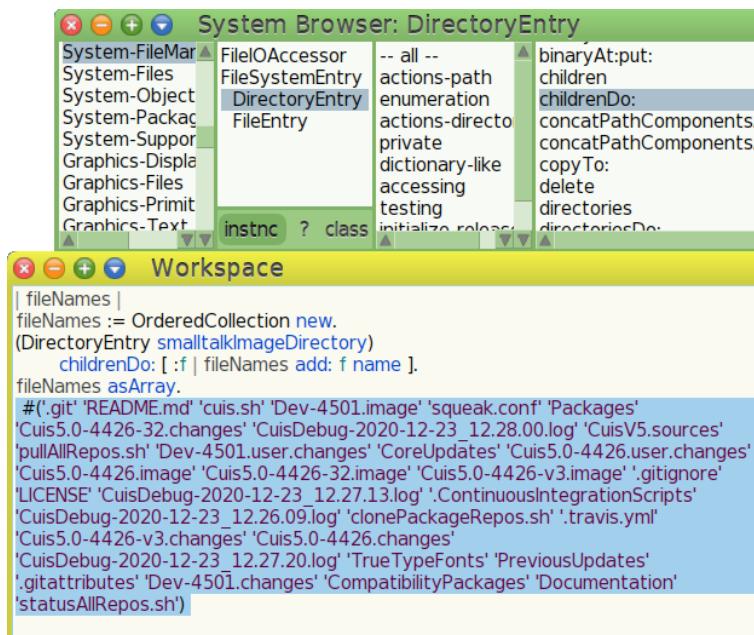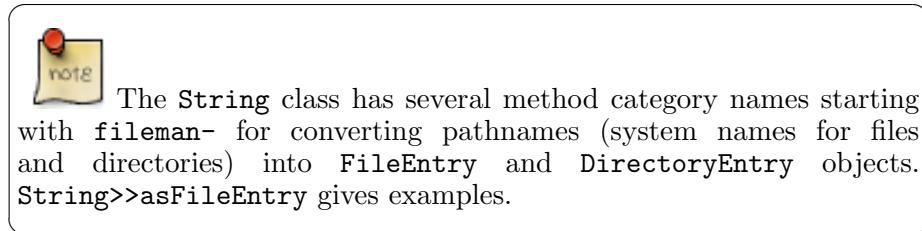
Figure 10.2: Names of files and directories in a Directory

> The `String` class has several method category names starting with `fileman-` for converting pathnames (system names for files and directories) into `FileEntry` and `DirectoryEntry` objects. `String>>asFileEntry` gives examples.

Now that we know what to expect, let us step through processing of the code using the debugger. Remove the result, then `Ctrl-a` (*select* **A***ll*) and `Ctrl-Shift-D` (*select* **D***ebug-it*).
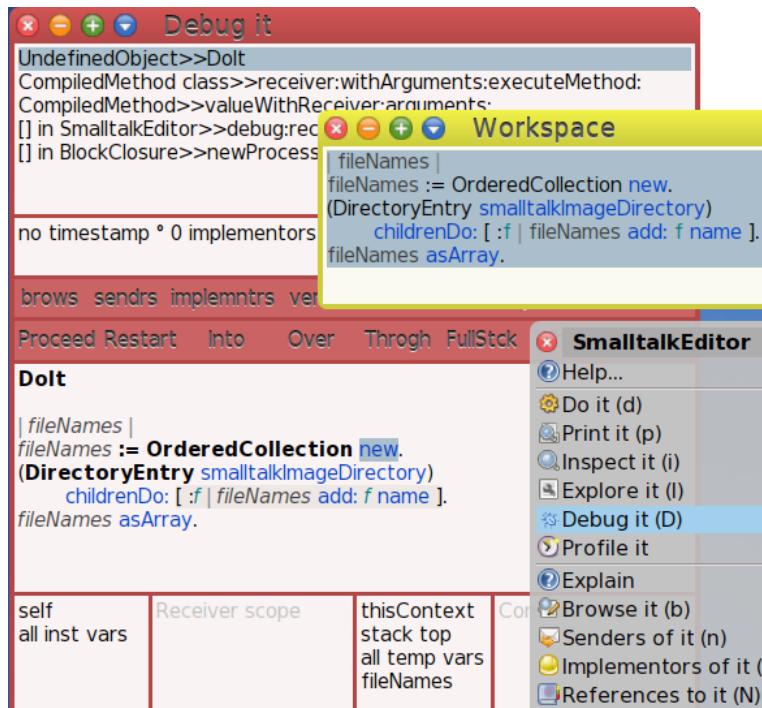
Figure 10.3: Debug It

The top pane in the debugger shows a view of the *execution stack* for this bit of execution context. The way to think of this, the *model of execution*, is that each time a method sends a message, it and its current state, arguments and local variables, are placed on a *stack* until the result of that message is received. If that message causes another message to be sent, then the new state is pushed onto the stack. When a result is returned, the *stack*

frame is *poped* and processing continues. This works like a stack of trays in a cafeteria.

The stack frames are displayed to show the stacked receiver and method. The focus object, the receiver, for the selected stack frame has an inspector in the lower left debugger panes at the bottom or the window.

The next two lower panes are an inspector for the arguments and local variables, or *temporaries*, of the context frame.

The larger area displays the code being processed and hilights the next message to be sent.

The stack of (framed) execution contexts gives a history of the computation so far. You can select any frame, view instance values in the receiver, view the arguments and method variables at that point.

The two rows of buttons above the code pane give additional views and control of how the execution processing is to proceed.

Notable buttons in the second row:

- **Proceed.** Continue execution

- **Restart.** Start execution of the current method from the beginning. You can edit a method shown in the code pane, save it, and restart it!

- **Into.** Step Into the code of the next message send.

- **Over.** Step Over the message send.
  Do the next message send, but stay in the current method.

- **Through.** Step into a block of code by going through – skipping – the intermediate message sends.
  Useful when you need to examine what is going on in a block of code, argument of the stepped message, for example the `#do:` message.

Now, we are going to play a bit. If you get out of synch with the instructions here, just close the debugger and start with Debug-It again.
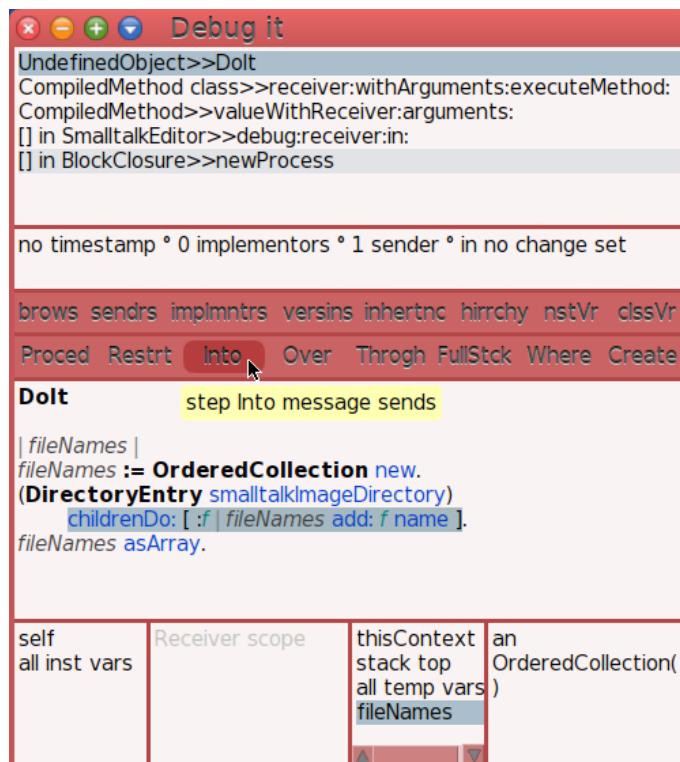
Figure 10.4: Step Into

As you *single step* the debugger, hilighting of the *next* message send changes. Press `Over` three times. You should see the line starting with *childrenDo:* hilighted. Now press `Into`.
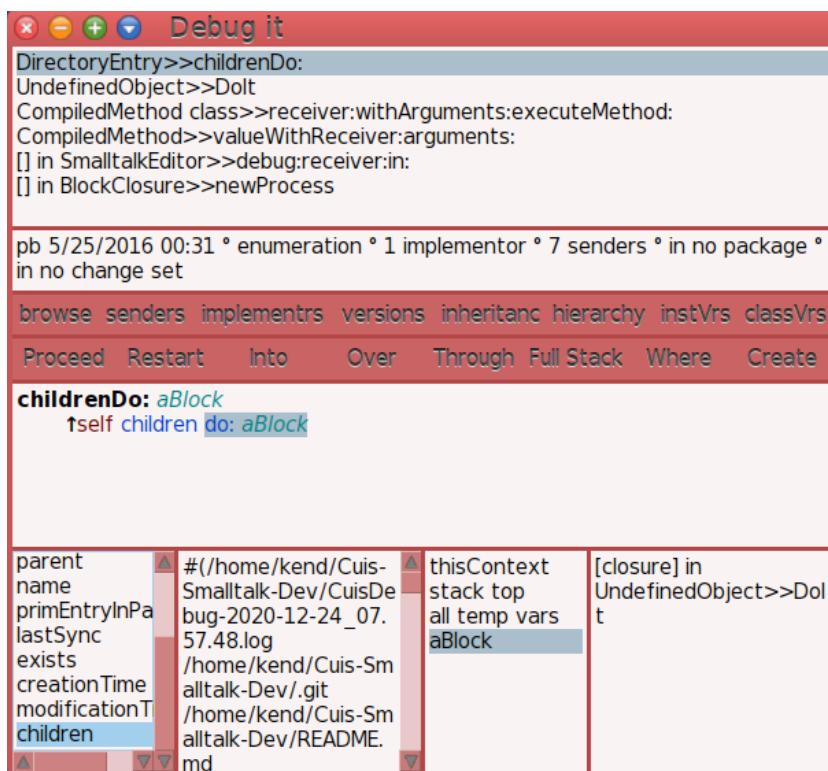
Figure 10.5: Viewing Focus Object and Temporaries

The stack area shows the focus object is a `DirectoryEntry`. Inspect its instance values by selecting lines in the lower left pane.

The stack area shows the focus method is `DirectoryEntry>>childrenDo:`. This is the method displayed in the code pane.

The argument to `childrenDo:` is `aBlock`. There are no method variables to display.

If you press *Over* again and *Into*, you should see the context where `do:` is being processed.

This might be a good place to investigate the inspectors, look up and down the stack, and play around a bit. By this time you should feel confident that you understand the basics of what is displayed here.

You are in control!

Let's look briefly at another way of doing this.

10.3 Halt!

A *breakpoint* is a place in code where one wishes to pause code processing and look around. One does not always want to single step to find a problem, especially one that occurs only once in a while. A breakpoint set where the problem occurs is quite handy.

In Smalltalk, one uses the `halt` method to set a breakpoint. The message `#halt` is sent to an object which is the debugger's initial focus.

Please change the Workspace code to add a `#halt` as follows.

```
| fileNames |
fileNames ← OrderedCollection new.
(DirectoryEntry smalltalkImageDirectory)
    childrenDo: [ :f | fileNames add: f name. fileNames halt ].
fileNames asArray.
```

Example 10.4: Halt: Set a Breakpoint

 The object which receives the `#halt` message becomes the focus object of the debugger.

Let's again *Ctrl-a* (*select* **A**ll) and *Ctrl-p* (*select* **P**rint-it).
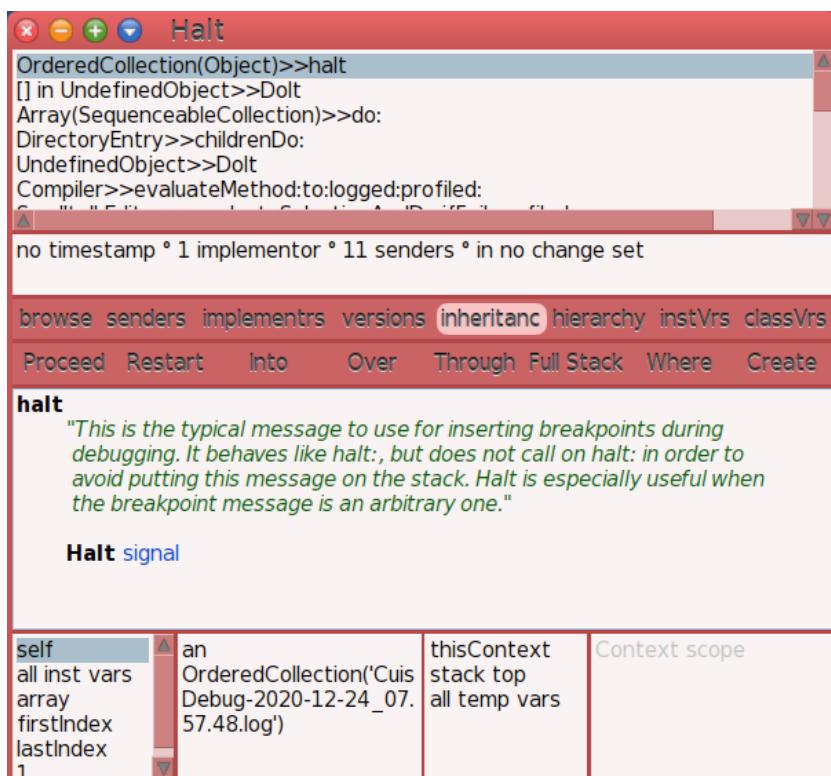
Figure 10.6: Halt

Press **Over** twice to step over the breakpoint.
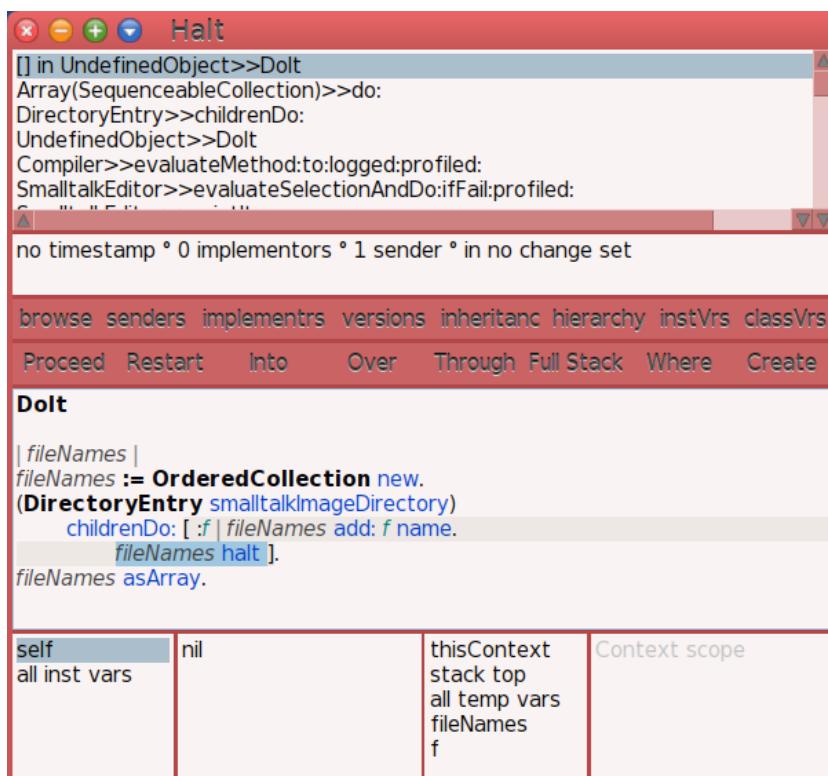
Figure 10.7: Step Over Breakpoint

Well, this looks familiar. I know what to do here.

Note that the `halt` is inside a loop. While in the loop, each time you press *Proceed* you will *hit* the breakpoint in the next iteration of the loop.

In many programming envionments, to make a change you need to kill a process, recompile code, then start a new process.

Smalltalk is a *live* environment. You can break, then change or write code (the *Create* button at mid-right), restart the stack frame, and continue processing – all without unwinding the context stack!

As an analogy, in many programming languages, it is like you stub your toe and visit a physician. The doctor says "Yes. you stubbed your toe." then takes out a gun, shoots you, and sends your mother a note "have another child." Smalltalk is much more friendly!

Note that with great power comes great responsibility.[1] In an open system, you can place a breakpoint anywhere, including places which can break the user interface! For example, it could be a bad thing to put a breakpoint in the code for the Debugger!

---

[1] https://quoteinvestigator.com/2015/07/23/great-power/

11 Sharing Cuis

> Programming is hardly ever a solitary communion between one man and one machine. Caring about other people is a conscious decision, and one that requires practice.
> —*Kent Beck, "Smalltalk Best Practice Patterns"(1997)*

> Programming remains an intensively collaborative process between groups of program readers and writers.
> —*Dave Thomas, "Smalltalk With Style"(1996)*

> Let your code talk — Names matter. Let the code say what it means. Introduce a method for everything that needs to be done. Dont be afraid to delegate, even to yourself.
> —*Oscar Nierstrasz, "Best Practice Patterns - talk slides"(2009)*

This book is an invitation.

We hope that you are using Cuis-Smalltalk to discover pathways of interest and are enjoying the journey. If so, at some point you have done something wonderful and probably want to share it.

Sharing requires communicating intent.

Good writing takes practice. Good writers read.

11.1 Golden Rules of the Smalltalk Guild

Basic questions, that appear to be the golden rules of the Smalltalk intergalactic guild[1]:

- Are methods short and understandable?
- Does a line of code read like a sentence?
- Do method names say what they do, rather than how they do it?
- Do class and instance variable names indicate their role(s)?
- Are there useful class comments?
- Can we make something simpler? Leave something out?

Now, we have been doing Smalltalk code for a while and so tend to follow good practices, but let's take another look at our code and see if we can make it easier for a reader to understand.

---

[1] *Don't panic*, at this stage of the book, the authors are still looking to find all the questions that really matter.

11.2 Refactoring to Improve Understanding

Browsing through the code, I note a method named **#left**, which seems perhaps like an abbreviation. I can ask for *senders* to see how **#left** is used in code elsewhere.
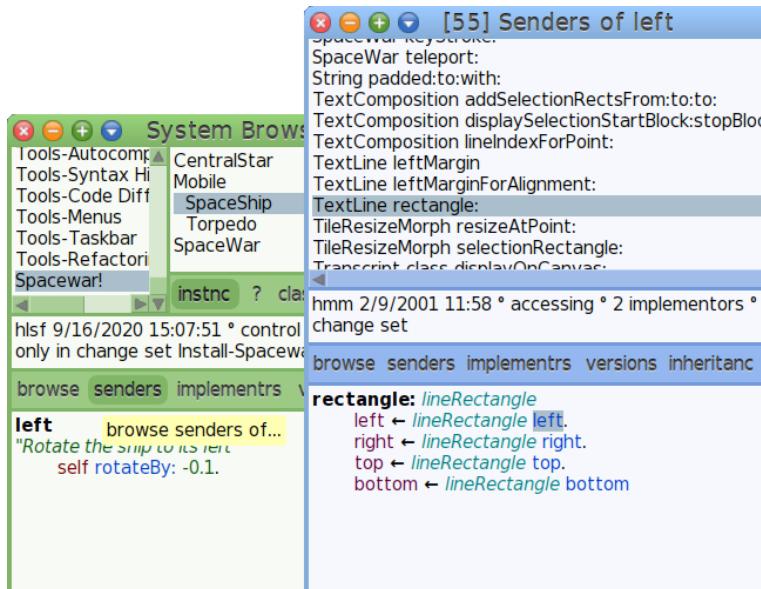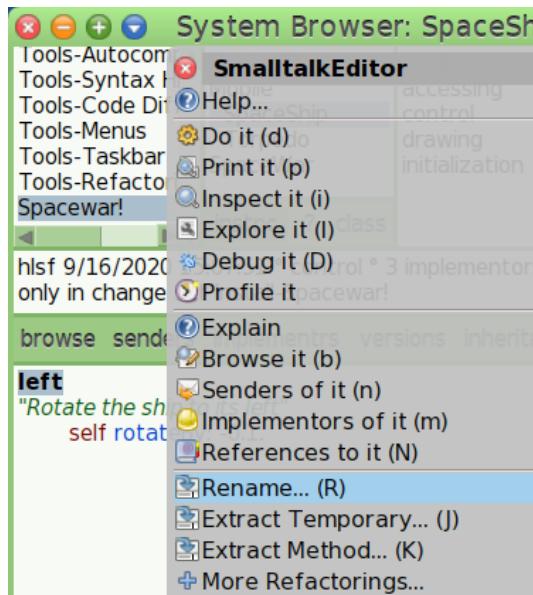
Figure 11.1: Senders of **left**

I notice that most uses of **#left** are to indicate a position, not take an action. How can I fix that?

Because people frequently want to change things for the better, there are a number of handy tools to help do this.

Now I could look at our uses of **#left** in Spacewars!, but the Cuis IDE already knows how to do this!

If I right-click on the Method Pane in the Browser, I get a context menu with selections to help me out. Here I choose **Rename**.

Figure 11.2: Rename `left`

Now the tools that help us refactor code are quite powerful, so restraint is called for. I don't want to change all uses of `#left` in the Cuis-Smalltalk system, just in the `Spacewar!` category.
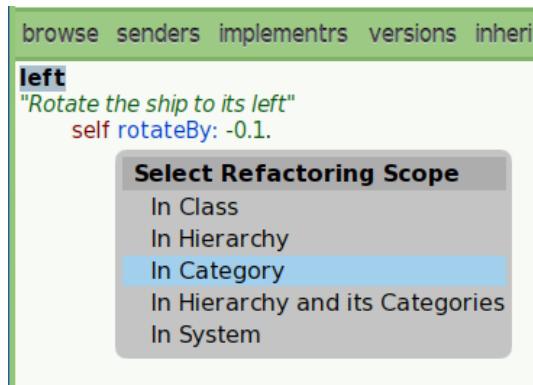
Figure 11.3: Rename in Category

Of course, when making changes one wants to see that the result is what one expects.
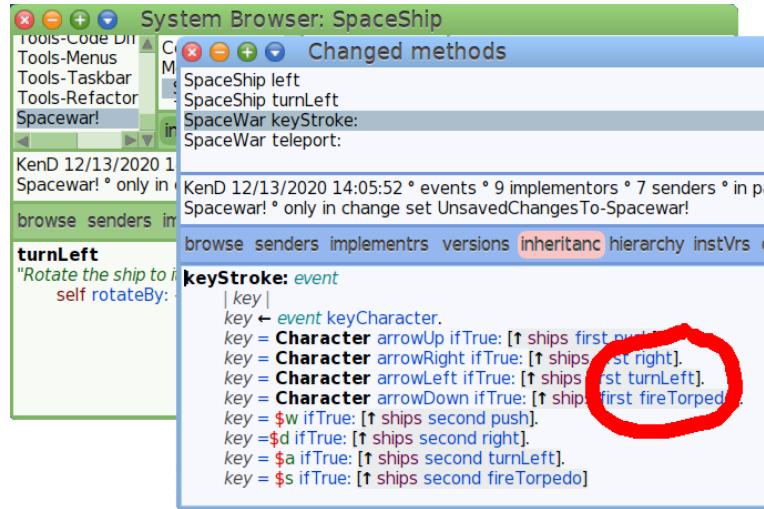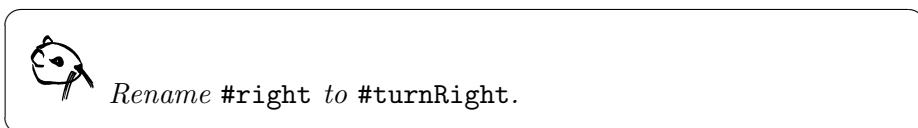
Figure 11.4: Results of Renaming

As I am not perfect, I tend to save the Cuis-Smalltalk image before I make large changes using powerful tools. If something happens that I did not want, I can then quit the image without saving and restart the saved image which remembers the world before I made the change.

Rename #right *to* #turnRight.

Exercise 11.1: Renaming a method

Looking around some more in the Browser, I notice the method `SpaceShip>>nose`.

Where did I use this? Ah, senders..
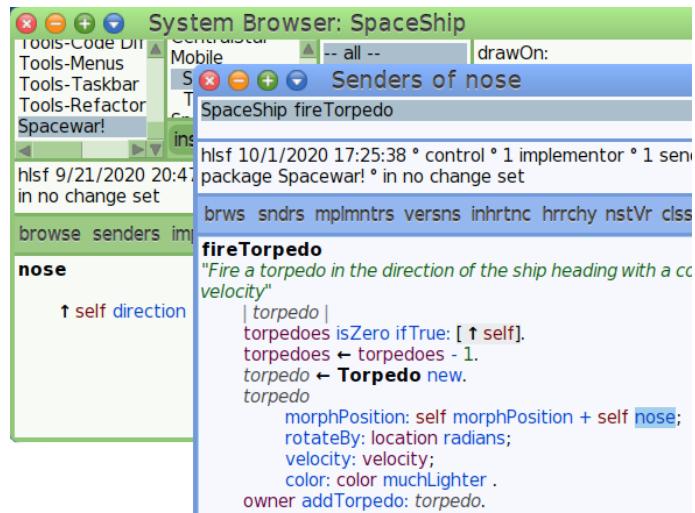
Figure 11.5: Senders of `nose`

Hmmm, perhaps something more specific. How about `#noseDirection`? How does that look?
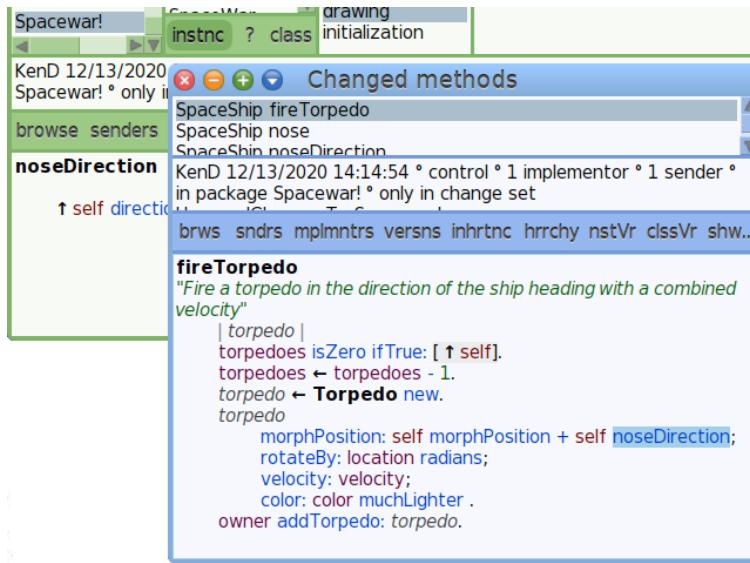
Figure 11.6: Rename `nose` to `noseDirection`

> World menu → `Help` is your friend. The `Terse Guide to Cuise` gives access to a large sample of code usages. The `Class Comment Browser` is an alternate way to find interesting class information. There are also more notes on code managenent and how we use GitHub.

> We want to share with you! Please visit packages at the main Cuis-Smalltalk repository at `https://github.com/Cuis-Smalltalk`, search GitHub for repositories with names starting with **Cuis-Smalltalk-**, and perhaps take a look at tutorials and information in `https://github.com/Cuis-Smalltalk/Learning-Cuis`.

There is much more to explore, but this book is an *introduction* and we have to stop writing text somewhere. This is a good place. We want to get back to writing code! And we look forward to seeing *your* projects!

Welcome to Cuis-Smalltalk!

Appendix A Documents Copyright

Part of the syntax chapter from the *Squeak by Example* book was borrowed and edited in the present book.
Copyright © 2007, 2008, 2009 by Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz and Damien Pollet.

Figure 1.4

Spacewar running on PDP-1, Joi Ito, 12 May 2007, resized,
https://www.flickr.com/photos/35034362831@N01/494431001
https://creativecommons.org/licenses/by/2.0/deed.en

[Adele Goldberg quote], page 31

Oral History of Adele Goldberg, Computer History Museum, 10 May 2010

Cuis-Smalltalk mascot

The southern mountain cavy (Microcavia australis) is a species of South American rodent in the family Caviidae.
Copyright © Euan Mee

Appendix B Summary of Syntax

Syntax	What it represents
`startPoint`	a variable name
`Transcript`	a global variable name
`self`	pseudo-variable
`1`	decimal integer
`2r101`	binary integer (radix 2)
`16r1a`	hexadecimal integer (radix 16)
`1.5`	floating point number
`2.4e7`	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
`|xy|`	declaration of variables x and y
`x ← 1, x := 1`	assign 1 to x
`[x + y]`	a block that evaluates to x+y
`<primitive: 1>`	virtual machine primitive or annotation
`3 factorial`	a unary message
`3+4`	a binary message
`2 raisedTo: 6 modulo: 10`	a keyword message
`↑ true, ^ true`	return the value true
`Transcript show: 'hello'.` `Transcript cr`	expression separator (.)
`Transcript show: 'hello'; cr`	message cascade (;)
`BoxedMorph new :: color: Color` `blue; openInWorld`	message cascade (::)
`` `{ 3@4 . 56 . 'click me'}` ``	the compound literal `#(3@4 56 'click me')`

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 argument, 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. `Smalltalk` is a global variable, an instance of the class `SystemDictionary`.

The receiver.

`self` is a keyword that refers to the object inside which the current method is executing. We call it "the receiver" because this object will normally have received the message that caused the method to execute. self is called a "pseudo-variable" since we cannot assign to it.

Integers.

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

Float point numbers.

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

Characters.

A dollar sign introduces a literal character: `$a` is the literal for 'a'. Instances of 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 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'`.

Static arrays.

Static arrays or Compile-time arrays are defined by `#( )`, surrounding space-separated literals. Everything within the parentheses must be a compile-time constant. For example, `#(27 #(true false) abc)` is a literal array of three elements: the integer `27`, the compile-time array containing the two booleans, and the symbol `#abc`.

Dynamic arrays.

Dynamic arrays or Run-time arrays. Curly braces { } define a (dynamic) array at run-time. Elements are expressions separated by periods. So `{ 1. 2. 1+2 }` defines an array with elements `1`, `2`, and the result of evaluating `1+2`. (The curly-brace notation is peculiar to the Squeak family dialect of Smalltalk! In other Smalltalks you must build up dynamic arrays explicitly.)

Comments.

Comments are enclosed in double quotes. `"hello"` is a comment, not a string, and is ignored by the Cuis-Smalltalk compiler. Comments may span multiple lines.

Local variable declarations.

Vertical bars `| |` enclose the declaration of one or more local variables in a method (and also in a block).

Assignment.

`:=` assigns an object to a variable. In the printed version of the book we wrote ← instead. Since this character is not present in the keyboard, you key in with the underscore character key. So, `x := 1` is the same as `x ← 1` or `x _ 1`.

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 and can have local variables.

Primitives.

`<primitive: ...>` denotes an invocation of a virtual machine primitive. (`<primitive: 1>` is the VM primitive for `SmallInteger>>+`.) Any code following the primitive is executed only if the primitive fails. The same syntax is also used for method annotations.

Unary messages.

Unary messages consist of a single word (like `#factorial`) sent to a receiver (like 3).

Binary messages.

Binary messages are operators (like `+`) sent to a receiver and taking a single argument. In `3 + 4`, the receiver is `3` and the argument is `4`.

Keyword messages.

Keyword messages consist of multiple 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.

Method return.

↑ is used to return a value from a method. (You must type `^` to obtain the ↑ character.)

Sequences of statements.

A period or full-stop (.) is the statement separator. Putting a period between two expressions turns them into independent statements.

Cascades.

There are two kinds of message cascades.

Semicolons can be used to send a cascade of messages to the original receiver. In `Transcript show: 'hello'; cr` we first send the keyword message `#show: 'hello'` to the receiver `Transcript`, and then we send the unary message `#cr` to the same receiver.

It is sometimes useful to send messages to the *result* of a message send. In `BoxedMorph new :: color: Color blue; openInWorld.` we send suceeding messages to the new BoxedMorph instance, not the BoxedMorph class.

Compound Literal

Backticks (`` ` ``) can be used to create compound literals at compile time. All components of a compound literal must be known when the code is compiled.

Appendix C The Exercises

Appendix D Solutions to the Exercises

Preface

Exercise 1

In the seventies, four versions were developed: Smalltalk-71, Smalltalk-72, Smalltalk-76 and Smalltalk-80.

Smallltalk Philosophy

Exercise 1.1
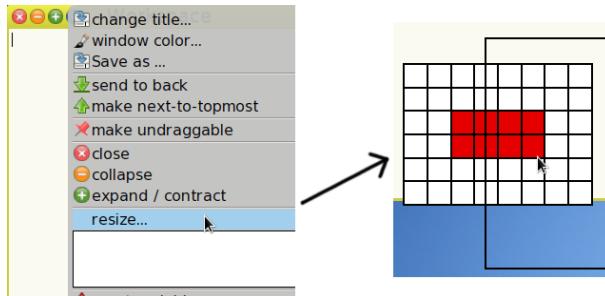
Figure D.1: Placement

Exercise 1.2

```
Transcript show: 'Hello ', 'my beloved ' asUppercase, 'friend'
```

Exercise 1.3

```
1 + (1/2) + (1/3) + (1/4)
⇒ 25/12
```

Exercise 1.4

Several messages can be sent one after the other:

```
Transcript show: 2020 printStringWords capitalized
```

The Message Way of Life

Exercise 2.1

```
'Hello'
   at: 1 put: $B;
```

```
    at: 2 put: $e;
    at: 3 put: $l;
    at: 4 put: $l;
    at: 5 put: $e;
    yourself
```

Exercise 2.2

```
1 + (1/2) squared + (1/3) squared + (1/4) squared
⇒  205 / 144
```

Exercise 2.3

From a System Browser, do from the left panel to the right ...`Kernel-Text` → `String` → `arithmetic`... the count of methods in the last right panel is 6: `*`, `+`, `-`, `/`, `//` and `\\`.

Class, model of Communicating Entities

Exercise 3.1

When the `Float` is selected, the wide text pane prints: "class definition for Float ° 92 instance methods ° 34 class methods ° 1280 total lines of code"

Exercise 3.2

```
0 to: Float twoPi by: 1/10 do: [:i |
    Transcript show: i cos; cr]
```

Exercise 3.3

1024 is not a random number. It is 2^{10} then written in base 2 : 10000000000, it is also `1 << 10`:

```
2↑10 ⇒ 1024
1024 printStringBase: 2 ⇒ '10000000000'
1 << 10 ⇒ 1024
```

Therefore, to multiply an integer by 1024, we shift left of 10 its digits:

```
360 << 10 ⇒ 368640
360 * 1024 ⇒ 368640
```

Exercise 3.4

```
5.2 + 0.9  - 6.1
⇒ 8.881784197001252e-16

5.2 + 0.7 + 0.11
⇒ 6.010000000000001

1.2 * 3  - 3.6
⇒ -4.440892098500626e-16
```

Exercise 3.5

The system returns the error `ZeroDivide`, division by zero.

Exercise 3.6

```
(52/10) + (9/10)  - (61/10)
⇒ 0

(52/10) + (7/10)  + (11/100)
⇒ 601/100 soit 6.01

(12/10) * 3  - (36/10)
⇒ 0
```

Exercise 3.7

There are different options, with slightly different results:

```
'There are 12 apples' select: [:i |i isLetter].
⇒ 'Thereareapples'
```

Not really satisfying. So another option:

```
'There are 12 apples' select: [:i |i isDigit not].
⇒ 'There are  apples'
```

Or even a shorter option with the `#reject:` message:

```
'There are 12 apples' reject: [:i |i isDigit].
⇒ 'There are  apples'
```

Exercise 3.8

In `String`, search for the method category `format`, there you find the `format:` method:

```
'Joe bought {1} apples and {2} oranges' format: #(5 4)
⇒  'Joe bought 5 apples and 4 oranges'
```

Exercise 3.9

The `SpaceWar`, `CentralStar` and `SpaceShip` definitions with their added instance variable should look like:

```
Object subclass: #SpaceWar
   instanceVariableNames: 'centralStar ships torpedoes'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

Object subclass: #CentralStar
   instanceVariableNames: 'mass'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

```
Object subclass: #SpaceShip
    instanceVariableNames: 'position heading velocity
        fuel torpedoes mass acceleration'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spacewar!'
```

Exercise 3.10

```
SpaceShip>>position
    ↑ position

SpaceShip>>velocity
    ↑ position

SpaceShip>>mass
    ↑ mass
```

Exercise 3.11

```
SpaceShip>>position: aPoint
    position ← aPoint

SpaceShip>>velocity: aPoint
    velocity ← aPoint

Torpedo>>heading: aFloat
    heading ← aFloat
```

Exercise 3.12

```
SpaceShip>>left
"Rotate the ship to its left"
    heading ← heading + 0.1

SpaceShip>>right
"Rotate the ship to its right"
    heading ← heading - 0.1
```

Exercise 3.13

```
SpaceShip>>push
"Init an acceleration boost"
    acceleration ← 10

SpaceShip>>unpush
"Stop the acceleration boost"
    acceleration ← 0
```

Exercise 3.14

```
CentralStar>>initialize
    super initialize.
    mass ← 8000.
```

The Collection Way of Life

Exercise 4.1

Open the protocol browser on the class `String`, search for the method `allButFirst:` implemented in `SequenceableCollection`. Read its comment in its source code.

```
'Hello My Friend' allButFirst: 6
⇒ 'My Friend'
```

Exercise 4.2

```
(-80 to: 50) asArray
```

Exercise 4.3

```
(1 to: 100) difference: (25 to: 75)
⇒ #(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95 96 97 98 99 100)
```

Exercise 4.4

```
(-20 to: 45) select: [:z | z odd]
```

Exercise 4.5

```
((101 to: 200) select: [:n | n isPrime]) size
⇒ 21
```

Exercise 4.6

```
(1 to: 100) select:[:n | n isDivisibleBy: 7]
⇒ #(7 14 21 28 35 42 49 56 63 70 77 84 91 98)
```

Exercise 4.7

This solution, based on set operations and multiple use of the `#select:` message, is mostly compatible with the knowledge acquired at this point of the book.

```
| primeNumbers nonPrimeNumbers |
primeNumbers ← (1 to: 100) select: [:n | n isPrime].
nonPrimeNumbers ← (1 to: 100) difference: primeNumbers.
nonPrimeNumbers select: [:n | n odd]
⇒ #(1 9 15 21 25 27 33 35 39 45 49 51 55 57 63 65 69 75
77 81 85 87 91 93 95 99)
```

A shorter solution with logical operations we have not discussed so far:

```
(1 to: 100) select:[:n | n isPrime not and: [n odd]]
```

Exercise 4.8

```
'Zpv!bsf!b!cptt' collect: [:c |
```

```
    (c asciiValue - 1) asCharacter]
⇒ 'You are a boss'
```

Exercise 4.9

```
($A to: $Z) collect: [:c |
    (c asciiValue - 65 + 3 \\ 26 + 65) asCharacter]
```

Each character from A to Z is referenced by the `c` block parameter and converted to its Ascii value[1]: `c asciiValue`. It is shifted of 65, so the A character counts as 0 and Z as 25 in the alphabet list. To apply the Caesar cipher, we send the message `+ 3` to the previous result.

We are almost good, only the characters X, Y, Z will overflow the alphabet, indeed these characters will be ciphered as 26, 27, 28. To fix that we send `\\ 26` to the previous result, it is an old trick of programming: the reminder of the Euclidean division by 26 frames the value between 0 and 25^2.

Finally we shift back of 65 before converting from the Ascii value to the character.

Exercise 4.10

In the solution of Exercise 4.9, we just need to replace the characters interval with a string:

```
'SMALLTALKEXPRESSION' collect: [:c |
    (c asciiValue - 65 + 3 \\ 26 + 65) asCharacter]
⇒ 'VPDOOWDONHASUHVVLRQ'
```

Exercise 4.11

```
'DOHDMDFWDHVW' collect: [:c |
    (c asciiValue - 65 - 3 \\ 26 + 65) asCharacter]
⇒ 'ALEAJACTAEST'
```

Exercise 4.12

The appropriate message is `#first:`, defined in the parent class `SequenceableCollection`. You need to use the protocol or hierarchy browser on `Array` to discover it:

```
array1 first: 2
⇒ #(2 'Apple')
```

Exercise 4.13

You could simply do a thumb:

```
array1 at: 1 put: 'kiwi'.
```

---

[1] https://en.wikipedia.org/wiki/ASCII

[2] It generalizes to the mathematics field of modular arithmetic, https://www.wikiwand.com/en/Modular_arithmetic

```
array1 at: 2 put: 'kiwi'.
array1 at: 3 put: 'kiwi'.
array1 at: 4 put: 'kiwi'.
```

Or even a bit less thumb:

```
1 to: array1 size do: [:index |
    array1 at: index put: 'kiwi']
```

But if you search for carefully the `Array` protocol, you can just do:

```
array1 atAllPut: 'kiwi'.
```

Exercise 4.14

In the `OrderedCollection` protocol search for the method `add:after:`.

```
coll1 ← {2 . 'Apple' . 2@1 . 1/3 } asOrderedCollection .
coll1 add: 'Orange' after: 'Apple'; yourself.
⇒  an OrderedCollection(2 'Apple' 'Orange' 2@1 1/3)
```

Exercise 4.15

```
Set  new
    addAll: 'buenos días';
    addAll: 'bonjour';
    yourself.
⇒  a Set($e $j $o $a $u $b $  $í $r $d $n $s)
```

Exercise 4.16

```
colors keysDo: [:key |
    colors at: key put: key asString capitalized].
colors
⇒ a Dictionary(#blue->'Blue' #green->'Green' #red->'Red'
#yellow->'Yellow' )
```

Exercise 4.17

When the game starts there is no fired torpedoes, therefore `torpedoes` is an empty `OrderedCollection`, instantiated with the `#new` class message.

In the other hand, the `ships` is an `Array` containing only two elements, the player ships. We use the `#with:with` class message to instantiate and populate the array with two ships created in the argument message.

For the readability, we split the code in several lines with the appropriate indentation.

```
torpedoes ← OrderedCollection new.
ships ← Array
    with: SpaceShip new
    with: SpaceShip new.
```

Exercise 4.18

```
SpaceWar>>stepAt: msSinceLast
```

```
ships do: [:each | each update: msSinceLast / 1000 ].
ships do: [:each | each unpush ].
torpedoes do: [:each | each update: msSinceLast / 1000 ].
```

Control Flow Messaging

Exercise 5.1

```
| divisors |
divisors ← [:x | (1 to: x) select: [:d | x \\ d = 0] ].
divisors value: 60.
⇒ #(1 2 3 4 5 6 10 12 15 20 30 60)
divisors value: 45
⇒ #(1 3 5 9 15 45)
```

Exercise 5.2

Check the implementations in `Boolean`, `True` and `False`.

Exercise 5.3

Once the method is edited and saved, in the `Method` pane select its name
`teleport:` then do ...right click → `more...` → `change category...` →
events...

Exercise 5.4

In the `Method` pane, select one uncategorized control method, then do ...right
click → `more...` → `change category` → `new...` key-in `control`.

To categorized the remaining uncategorized control methods, repeat but
select `control` at the last step as this category now exists.

Exercise 5.5

We do not need an iterator to detect a collision between two ships. However
we use an iterator to take action on each ship when a collision is detected.

```
SpaceWar>>collisionsShips
| positionA position B |
  positionA ← ships first morphPosition.
  positionB ← ships second  morphPosition.
  (positionA dist: positionB) < 25 ifTrue: [
     ships do: [:each |
        each flashWith: Color red.
        self teleport: each]
  ]
```

Local variables only used to ease the code source formatting in printed
book.

Exercise 5.6

You just need to pick the appropriate code snippets from the referenced exercise and examples.

```
SpaceWar>>collisionsTorpedoesStar
| position |
   position ← centralStar morphPosition.
   torpedoes do: [:each |
      (each morphPosition dist: position) < 8 ifTrue: [
         each flashWith: Color orange.
         self destroyTorpedo: each]]
```

Visual with Morph

Exercise 6.1

Just replace all `Object` occurrences with `PlacedMorph`:

```
PlacedMorph subclass: #SpaceWar
   instanceVariableNames: 'centralStar ships torpedoes'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

PlacedMorph subclass: #CentralStar
   instanceVariableNames: 'mass'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

PlacedMorph subclass: #SpaceShip
   instanceVariableNames: 'name position heading velocity
      fuel torpedoes mass acceleration'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Exercise 6.2

```
Mobile subclass: #SpaceShip
   instanceVariableNames: 'name fuel torpedoes'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

Mobile subclass: #Torpedo
   instanceVariableNames: 'lifeSpan'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

The Fundamentals of Morph

Exercise 7.1

The `drawOn:` method is modified to draw two distinct, unconnected lines:

```
LineExampleMorph>>drawOn: aCanvas
    aCanvas strokeWidth: 20 color: Color green do: [
        aCanvas
            moveTo: 0 @ 0;
            lineTo: 200 @ 200;
            moveTo: 200 @ 0;
            lineTo: 0 @ 200 ]
```

Learn how the `#moveTo:` message moves the pencil to a given position with the pen up, then the `#lineTo:` asks the pencil to draw from that previous position to this new position.

Exercise 7.2

We create a `RectangleExampleMorph`, subclass of `PlacedMorph`:

```
PlacedMorph subclass: #RectangleExampleMorph
    instanceVariableNames: 'fillColor'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

Then its necessary methods to initialize and to draw the morph:

```
initialize
    super initialize .
    fillColor ← Color random alpha: 0.5

drawOn: aCanvas
    aCanvas
        strokeWidth: 1
        color: Color blue
        fillColor: fillColor
        do: [
            aCanvas moveTo: 0 @ 0;
                lineTo: 200 @ 0;
                lineTo: 200 @ 100;
                lineTo: 0 @ 100;
                lineTo: 0 @ 0]
```
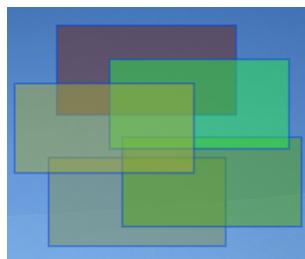
Figure D.2: Several rectangle morphs

Exercise 7.3

Among the clock parts (submorphs) we only need to modify the drawing of the `ClockSecondHandMorph` class. The disc is surrounded with a thin red line and filled in yellow.

```
ClockSecondHandMorph>>drawOn: aCanvas
   aCanvas strokeWidth: 1.5 color: Color red do: [
      aCanvas
         moveTo: 0 @ 0;
         lineTo: 85 @ 0 ].
   aCanvas ellipseCenter: 0 @ -70 radius: 3 @ 3
      borderWidth: 1
      borderColor: Color red fillColor: Color yellow
```

Exercise 7.4

The width of the torpedo is 4 pixels and its height 8 pixels:

```
Torpedo>>morphExtent
   ↑ `4 @ 8`
```

Exercise 7.5

The `Torpedo`'s `drawOn:` method is very similar to the one in `SpaceShip` class:

```
Torpedo>>drawOn: canvas
   | a b c |
   a ← 0 @ -4.
   b ← -2 @ 4.
   c ← 2 @ 4.
   canvas line: a to: b width: 2 color: color.
   canvas line: c to: b width: 2 color: color.
   canvas line: a to: c width: 2 color: color.
```

Exercise 7.6

We use a local variable because we use two times the vertices, one to draw the ship and a second time to draw the gas exhaust.

```
SpaceShip>>drawOn: canvas
   | vertices  |
   vertices ← self class vertices.
   canvas line: vertices first to: vertices second width: 2 color: color.
   canvas line: vertices second to: vertices third width: 2 color: color.
   canvas line: vertices third to: vertices fourth width: 2 color: color.
   canvas line: vertices fourth to: vertices first width: 2 color: color.
   "Draw gas exhaust"
   acceleration ifNotZero: [
      canvas line: vertices third to: 0@35 width: 1 color: Color gray]
```

Exercise 7.7

You need both to iterate each vertex of the `vertices` array and access the subsequent vertex by index. The arithmetic reminder operation `#\\` is needed to keep the index in the boundary of the collection.

When `size` is 4 (Space ship diagram), the argument (i \\ size + 1) takes alternatively the following values:

- **i = 1** => 1 \\ 4 + 1 = 1 + 1 = **2**
- **i = 2** => 2 \\ 4 + 1 = 2 + 1 = **3**
- **i = 3** => 3 \\ 4 + 1 = 3 + 1 = **4**
- **i = 4** => 4 \\ 4 + 1 = 0 + 1 = **1**

```
Mobile>>drawOn: canvas polygon: vertices
   | size |
   size ← vertices size.
   vertices withIndexDo: [: aPoint :i |
      canvas
         line: aPoint
         to: ( vertices at: (i  \\ size + 1) )
         width: 2
         color: color]
```

Exercise 7.8

Just replace each morph position distance approach with the intersection detection between the morphs' display bounds:

```
SpaceWar>>collisionsShips
   (ships first collides: ships second)
   ../..

SpaceWar>>collisionsShipsTorpedoes
   ships do: [:aShip |
      torpedoes do: [:aTorpedo |
         (aShip collides: aTorpedo)
```

```
../..

SpaceWar>>collisionsTorpedoesStar
    torpedoes do: [:each |
       (each collides: centralStar)
    ../..
```

Events

Exercise 8.1

The method `handlesMouseOver:`, implemented in the `SpaceWar` morph class, returns true so the game play is informed of mouse over events in dedicated methods.

```
SpaceWar>>handlesMouseOver: event
    ↑ true
```

Exercise 8.2

You need to browse the `Morph>>handlesMouseOver:` method and read the comment. It writes about a `#mouseEnter:` message; we implement the matching method in `SpaceWar` class with the behaviors previously described:

```
SpaceWar>>mouseEnter: event
    event hand newKeyboardFocus: self.
    self startStepping
```

Exercise 8.3

The message `#mouseLeave:` is sent to our `SpaceWar` instance each time the mouse cursor move out (leaves) of the game play. Therefore we add the homonym method to the `SpaceWar` class:

```
SpaceWar>>mouseLeave: event
    event hand releaseKeyboardFocus: self.
    self stopStepping
```

Exercise 8.4

The `#handlesKeyboard` message is sent to a morph to know if it wants to receive keyboard event. The morph responds true to this message to state its interest on keyboard event. We implement the method in the `SpaceWar` class:

```
SpaceWar>>handlesKeyboard
    ↑ true
```

Exercise 8.5

We designate the characters as $w $a $s $d. We append the code below to the method `SpaceWar>>keyStroke`:

```
key = $w ifTrue: [↑ ships second push].
```

```
key = $d ifTrue: [↑ ships second right].
key = $a ifTrue: [↑ ships second left].
key = $s ifTrue: [↑ ships second fireTorpedo]
```

Code Management

Exercise 9.3

1. In the System Browser, create two class categories: in its most left pane menu select `add item... (a)` and key in one category name at a time.
2. Create two classes, in each category: `TheCuisBook` subclass of `Object` and `TheBookMorph` subclass of `PlacedMorph`.

   The two operations above are doable in one shot. Select any existing category and key in the class definition with the category name:

   ```
   PlacedMorph subclass: #TheBookMorph
       instanceVariableNames: ''
       classVariableNames: ''
       poolDictionaries: ''
       category: 'TheCuisBook-Views'
   ```

   Once you save the class definition, the `TheBookMorph` class is created – obviously – but also the `TheCuisBook-Views` category!
3. Go to the Installed Packages tool ...World menu → `Open...` → `Installed Packages`..., press the `new` button and key in `TheCuisBook`.
4. Press the `save` button, you are done!

Appendix E The Examples

Appendix F The Figures

Appendix G Spacewar! Source Code

In the raw source code below, understand the "_" character as "←".

```
'From Cuis 6.0 [latest update: #5661] on 22 February 2023 at 9:33:46 pm'!
'Description SpaceWar!! game code'!
!provides: 'Spacewar!!' 1 92!
SystemOrganization addCategory: 'Spacewar!!'!


!classDefinition: #Mobile category: 'Spacewar!!'!
PlacedMorph subclass: #Mobile
        instanceVariableNames: 'acceleration color velocity'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Spacewar!!'!
!classDefinition: 'Mobile class' category: 'Spacewar!!'!
Mobile class
        instanceVariableNames: 'vertices'!

!classDefinition: #SpaceShip category: 'Spacewar!!'!
Mobile subclass: #SpaceShip
        instanceVariableNames: 'name fuel torpedoes'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Spacewar!!'!
!classDefinition: 'SpaceShip class' category: 'Spacewar!!'!
SpaceShip class
        instanceVariableNames: ''!

!classDefinition: #Torpedo category: 'Spacewar!!'!
Mobile subclass: #Torpedo
        instanceVariableNames: 'lifeSpan'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Spacewar!!'!
!classDefinition: 'Torpedo class' category: 'Spacewar!!'!
Torpedo class
        instanceVariableNames: ''!

!classDefinition: #SpaceWar category: 'Spacewar!!'!
PlacedMorph subclass: #SpaceWar
        instanceVariableNames: 'color ships torpedoes centralStar'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Spacewar!!'!
!classDefinition: 'SpaceWar class' category: 'Spacewar!!'!
SpaceWar class
        instanceVariableNames: ''!

!classDefinition: #CentralStar category: 'Spacewar!!'!
Morph subclass: #CentralStar
```

```
          instanceVariableNames: ''
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Spacewar!!'!
!classDefinition: 'CentralStar class' category: 'Spacewar!!'!
CentralStar class
          instanceVariableNames: ''!



!Mobile commentStamp: 'hlsf 10/25/2020 09:16:19' prior: 0!
An abstract mobile with a mass and inner acceleration, subject to a gravity pull.!

!SpaceShip commentStamp: 'hlsf 10/25/2020 09:19:28' prior: 0!
A ship comes with a torpedo inventory and a fuel tank.
Player controls its heading and acceleration, and he can fire torpedo.!

!Torpedo commentStamp: 'hlsf 10/25/2020 09:18:37' prior: 0!
A torpedo comes with a lifespan and an automatic acceleration handling!

!SpaceWar commentStamp: 'hlsf 11/1/2020 10:06:46' prior: 0!
I am the view and controler of the game.
My submorphs are the elements of the game.

To start the game execute: "SpaceWar new openInWorld"

Ship Controls
Green: left, right, up down arrows
Red: a, d, w, s keys!

!CentralStar commentStamp: '<historical>' prior: 0!
I am the central start of the game generating a gravity field.!

!Mobile methodsFor: 'accessing' stamp: 'hlsf 10/1/2020 08:34:04'!
color: aColor
          "Set the receiver's color. "
          color = aColor ifFalse: [
                    color _ aColor.
                    self redrawNeeded ]! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 2/14/2022 13:48:54'!
direction
"I am an unit vector representing the noze direction of the mobile"
          ^Point rho: 1 theta: self heading! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 2/14/2022 13:48:44'!
heading
          ^ location radians - Float halfPi! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 2/14/2022 13:53:10'!
heading: aHeading
          self rotation: aHeading + Float halfPi! !
```

```
!Mobile methodsFor: 'accessing' stamp: 'hlsf 10/1/2020 08:34:23'!
mass
          ^ 1! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 8/23/2021 11:55:02'!
velocity: aVector
          velocity _ aVector ! !

!Mobile methodsFor: 'computing' stamp: 'hlsf 10/1/2020 08:34:37'!
gravity
"Compute the gravity acceleration vector"
          | position |
          position _ self morphPosition.
          ^ [-10 * self mass * owner starMass / (position r raisedTo: 3) * position]
                    on: Error do: [0 @ 0]! !

!Mobile methodsFor: 'computing' stamp: 'hlsf 8/23/2021 11:52:21'!
update: t
"Update the mobile position and velocity "
          | ai ag newVelocity |
          "acceleration vectors"
          ai _ acceleration * self direction.
          ag _ self gravity.
          newVelocity _ (ai + ag) * t + velocity.
          self morphPosition: (0.5 * (ai + ag) * t squared)
                    + (velocity * t)
                    + self morphPosition.
          velocity _ newVelocity.

          "Are we out of screen?
          If so we move the mobile to the other corner and slow it down by a factor of 2"
          (self isInOuterSpace and: [self isGoingOuterSpace]) ifTrue: [
                    velocity _ velocity / 2.
                    self morphPosition: self morphPosition negated]! !

!Mobile methodsFor: 'initialization' stamp: 'hlsf 10/1/2020 08:37:29'!
initialize
          super initialize.
          color _ Color gray.
          velocity _ 0 @ 0.
          acceleration _ 0.! !

!Mobile methodsFor: 'testing' stamp: 'hlsf 10/8/2020 22:21:33'!
isGoingOuterSpace
"is the mobile going crazy in direction of the outer space?"
          ^ (self morphPosition dotProduct: velocity) > 0 ! !

!Mobile methodsFor: 'testing' stamp: 'jmv 10/28/2021 15:04:22'!
isInOuterSpace
"Is the mobile located in the outer space? (outside of the game play area)"
          ^(owner morphLocalBounds containsPoint: self morphPosition) not! !
```

```
!Mobile methodsFor: 'drawing' stamp: 'hlsf 8/23/2021 11:50:43'!
drawOn: canvas polygon: vertices
          | size |
          size _ vertices size.
          vertices withIndexDo: [: aPoint :i |
                    canvas
                              line: aPoint
                              to: ( vertices at: (i  \\ size + 1) )
                              width: 2
                              color: color] ! !

!Mobile methodsFor: 'geometry' stamp: 'jmv 10/28/2021 14:53:01'!
morphLocalBounds

          ^Rectangle encompassing: self class vertices! !

!Mobile class methodsFor: 'as yet unclassified' stamp: 'hlsf 12/19/2020 23:21:09'!
vertices
          ^ vertices ! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 2/14/2022 13:53:33'!
fireTorpedo
"Fire a torpedo in the direction of the ship heading with a combined velocity"
          | torpedo |
          torpedoes isZero ifTrue: [ ^ self].
          torpedoes _ torpedoes - 1.
          torpedo _ Torpedo new.
          torpedo
                    morphPosition: self morphPosition + self nose;
                    heading: self heading;
                    velocity: velocity;
                    color: color muchLighter .
          owner addTorpedo: torpedo.! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 2/14/2022 14:09:51'!
left
"Rotate the ship to its left"
          self heading: self heading - 0.1! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 10/1/2020 17:26:59'!
push
"Init an accelaration boost"
          fuel isZero ifTrue: [^ self].
          fuel _ fuel - 1.
          acceleration _ 50! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 2/14/2022 14:10:08'!
right
"Rotate the ship to its right"
          self heading: self heading + 0.1! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 9/16/2020 15:08:18'!
```

```
unpush
"Stop the accelaration boost"
          acceleration _ 0! !


!SpaceShip methodsFor: 'drawing' stamp: 'hlsf 12/20/2020 15:18:30'!
drawOn: canvas
          | vertices  |
          vertices _ self class vertices.
          super drawOn: canvas polygon: vertices.
          "Draw gas exhaust"
          acceleration ifNotZero: [
                      canvas line: vertices third to: 0@35 width: 1 color: Color gray].! !


!SpaceShip methodsFor: 'drawing' stamp: 'hlsf 12/15/2020 21:16:09'!
nose
          ^ self direction * 40! !


!SpaceShip methodsFor: 'initialization' stamp: 'hlsf 2/14/2022 13:50:54'!
initialize
          super initialize.
          self resupply! !


!SpaceShip methodsFor: 'initialization' stamp: 'hlsf 2/16/2023 11:54:41'!
resupply
          fuel _ 500.
          torpedoes _ 10! !


!SpaceShip class methodsFor: 'as yet unclassified' stamp: 'hlsf 12/19/2020 23:21:16'!
initialize
"SpaceShip initialize"
          vertices _ {0@-15 . -10@15. 0@10. 10@15}! !


!Torpedo methodsFor: 'computing' stamp: 'hlsf 2/16/2023 11:56:14'!
update: t
"Update the torpedo position"
          super update: t.
          "orientate the torpedo in its velocity direction, nicer effect while inaccurate"
          self heading: (velocity y arcTan: velocity x).
          lifeSpan _ lifeSpan - 1.
          lifeSpan isZero ifTrue: [owner destroyTorpedo: self].
          acceleration > 0 ifTrue: [acceleration _ acceleration - 500].! !


!Torpedo methodsFor: 'drawing' stamp: 'hlsf 12/20/2020 13:19:16'!
drawOn: canvas
          self drawOn: canvas polygon: self class vertices! !


!Torpedo methodsFor: 'initialization' stamp: 'hlsf 12/15/2020 21:12:29'!
initialize
          super initialize.
          lifeSpan _ 500.
          acceleration _ 3000! !
```

```
!Torpedo class methodsFor: 'as yet unclassified' stamp: 'hlsf 12/19/2020 23:21:22'!
initialize
"Torpedo initialize"
         vertices _ {0@-4 . -2@4 . 2@4}! !


!SpaceWar methodsFor: 'accessing' stamp: 'hlsf 9/16/2020 16:41:40'!
starMass
         ^ centralStar mass! !


!SpaceWar methodsFor: 'actions' stamp: 'hlsf 9/21/2020 20:56:33'!
addTorpedo: aTorpedo
         torpedoes add: aTorpedo.
         self addMorph: aTorpedo ! !


!SpaceWar methodsFor: 'actions' stamp: 'hlsf 9/21/2020 21:10:33'!
destroyTorpedo: aTorpedo
         aTorpedo delete.
         torpedoes remove: aTorpedo ! !


!SpaceWar methodsFor: 'actions' stamp: 'hlsf 10/1/2020 17:29:37'!
teleport: aShip
"Teleport a ship at a random location"
         | area randomCoordinate |
         aShip resupply.
         area _ self morphLocalBounds insetBy: 20.
         randomCoordinate _ [(area left to: area right) atRandom].
         aShip
                  velocity: 0@0;
                  morphPosition: randomCoordinate value @ randomCoordinate value! !


!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 10/1/2020 15:04:27'!
collisions
         self collisionsShipsStar.
         self collisionsTorpedoesStar.
         self collisionsShipsTorpedoes.
         self collisionsShips ! !


!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:53:43'!
collisionsShips
         (ships first collides: ships second) ifTrue: [
                  ships do: [:each |
                           each flashWith: Color red.
                           self teleport: each] ]! !


!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:28:09'!
collisionsShipsStar
         ships do: [ :aShip |
                  (aShip collides: centralStar) ifTrue: [
                           aShip flashWith: Color red.
                           self teleport: aShip ]]! !


!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:30:08'!
```

```
collisionsShipsTorpedoes
          ships do: [:aShip |
                    torpedoes do: [:aTorpedo |
                    (aShip collides: aTorpedo) ifTrue: [
                              aShip flashWith: Color red.
                              aTorpedo flashWith: Color orange.
                              self destroyTorpedo: aTorpedo.
                              self teleport: aShip]]
          ]! !

!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:30:19'!
collisionsTorpedoesStar
          torpedoes do: [:each |
                    (each collides: centralStar) ifTrue: [
                              each flashWith: Color orange.
                              self destroyTorpedo: each]]! !

!SpaceWar methodsFor: 'drawing' stamp: 'hlsf 10/1/2020 15:45:54'!
drawOn: aCanvas
          aCanvas
                    fillRectangle: self morphLocalBounds
                    color: color! !

!SpaceWar methodsFor: 'event handling testing' stamp: 'hlsf 11/1/2020 10:03:57'!
handlesKeyboard
            ^ true! !

!SpaceWar methodsFor: 'event handling testing' stamp: 'hlsf 11/1/2020 10:04:18'!
handlesMouseOver: event
            ^ true! !

!SpaceWar methodsFor: 'events' stamp: 'hlsf 2/22/2023 21:31:48'!
keyStroke: event
          | key |
          event isArrowUp ifTrue: [^ ships first push].
          event isArrowRight ifTrue: [^ ships first right].
          event isArrowLeft ifTrue: [^ ships first left].
          event isArrowDown ifTrue: [^ ships first fireTorpedo].
          key _ event keyCharacter.
          key = $w ifTrue: [^ ships second push].
          key =$d ifTrue: [^ ships second right].
          key = $a ifTrue: [^ ships second left].
          key = $s ifTrue: [^ ships second fireTorpedo]! !

!SpaceWar methodsFor: 'events' stamp: 'hlsf 9/11/2020 10:07:17'!
mouseEnter: evt
          evt hand newKeyboardFocus: self.
          self startStepping! !

!SpaceWar methodsFor: 'events' stamp: 'hlsf 9/11/2020 10:07:22'!
mouseLeave: evt
          evt hand releaseKeyboardFocus: self.
```

```
                   self stopStepping! !


!SpaceWar methodsFor: 'focus handling' stamp: 'hlsf 9/10/2020 16:04:35'!
keyboardFocusChange: gotFocus
          gotFocus
                   ifTrue: [color _ self defaultColor ]
                   ifFalse: [color _ self defaultColor alpha: 0.5].
          self redrawNeeded.! !


!SpaceWar methodsFor: 'geometry testing' stamp: 'hlsf 12/11/2020 13:00:49'!
clipsSubmorphs
          ^ true! !


!SpaceWar methodsFor: 'initialization' stamp: 'hlsf 9/10/2020 15:21:22'!
defaultColor
          ^ `Color black`! !


!SpaceWar methodsFor: 'initialization' stamp: 'hlsf 11/1/2020 10:04:39'!
initialize
"We want to capture keyboard and mouse events,
start the game loop(step) and initialize the actors."
          super initialize.
          color _ self defaultColor.
          self startSteppingStepTime: self stepTime.
          self initializeActors.! !


!SpaceWar methodsFor: 'initialization' stamp: 'hlsf 8/23/2021 11:53:32'!
initializeActors
          centralStar _ CentralStar new.
          self addMorph: centralStar.
          centralStar morphPosition: 0@0.
          torpedoes _ OrderedCollection new.
          ships _ Array with: SpaceShip new with: SpaceShip new.
          self addAllMorphs: ships.
          ships first
                   morphPosition: 200 @ -200;
                   color: Color green darker.
          ships second
                   morphPosition: -200 @ 200;
                   color: Color red darker.! !


!SpaceWar methodsFor: 'stepping' stamp: 'hlsf 12/16/2020 19:45:30'!
stepAt: millisecondSinceLast
          ships do: [:each | each update: millisecondSinceLast / 1000 ].
          ships do: [:each | each unpush ].
          torpedoes do: [:each | each update: millisecondSinceLast / 1000 ].
          self collisions.
          self redrawNeeded
! !


!SpaceWar methodsFor: 'stepping' stamp: 'hlsf 9/11/2020 13:19:07'!
stepTime
```

```
"millisecond"
          ^ 20! !

!SpaceWar methodsFor: 'stepping' stamp: 'hlsf 9/10/2020 18:03:59'!
wantsSteps
          ^ true! !

!SpaceWar methodsFor: 'geometry' stamp: 'hlsf 2/16/2023 11:38:27'!
morphLocalBounds

          ^ -300 @ -300 extent: 600@600! !

!CentralStar methodsFor: 'accessing' stamp: 'hlsf 10/8/2020 22:20:17'!
mass
          ^ 8000! !

!CentralStar methodsFor: 'accessing' stamp: 'hlsf 12/15/2020 21:04:13'!
morphExtent
          ^ `30 @ 30`! !

!CentralStar methodsFor: 'drawing' stamp: 'KenD 2/15/2023 15:40:04'!
drawOn: canvas
          | radius |
          radius _ self morphExtent // 2.
          canvas ellipseCenter: 0 @ 0
                    radius: (radius x + (2 atRandom - 1)) @ (radius y + (2 atRandom - 1))
                    borderWidth: 3
                    borderColor: Color orange
                    fillColor: Color yellow ! !

!CentralStar methodsFor: 'geometry' stamp: 'jmv 10/28/2021 14:55:55'!
morphLocalBounds

          ^Rectangle center: 0@0 extent: self morphExtent! !
SpaceShip initialize!
Torpedo initialize!
```

Appendix H Conceptual index

N

O

P

R