1

Methods: Named Message Sequences

Up to now, you used scripts: you created a robot and sent a sequence of messages to it. Using scripts is a straightforward approach, but it has some severe limitations. One of the major limitations is that a script cannot be called by another script. This is a real problem because a script cannot be reused by other scripts, you have to rewrite it again and again.

Wouldn't it be nice if one could define a kind of script whose sequence of messages could be sent to any robot? Well, this is actually possible, and such a sequence of messages is called a *method*[1]. A *method* is a named script. This name can be used in a script or even in another method. In fact, there is nothing really new here: All the robot messages you have used so far are methods that you used with any robot!

In this chapter, you shall learn how to define methods. You already know most of what you need to write the code of the methods. However a method must be defined using a special editor called a code browser. We start by comparing a script and a method. Then we will define a method, and finally step back and really look in detail at what we did.

1 Scripts versus Methods

Let's look at one of the scripts you have already written, for example script 1.1 that created a robot and asked it to draw a square 100 pixels wide.

Script 1.1 (*A simple square)*

```
| pica |
pica := Bot new.
4 timesRepeat:
      [ pica turnLeft: 90.
      pica go: 100 ]
```

The problem with this script is that each time you need to draw a square you need to *copy* the 3 last lines of Script 1.1. Also if you want another robot (for example daly) to draw the square, you must change the name pica to daly everywhere. This is illustrated by Script 1.4.

[1] In the context of this book we will not go into the full power of methods, as it involves object-oriented programming.

Script 1.2 (*Two simple squares)*

```
| pica daly |
pica := Bot new.
daly := Bot new.
daly jump: 200.
daly color: Color red.

4 timesRepeat:
      [ pica turnLeft: 90.
      pica go: 100 ].
4 timesRepeat:
      [ daly turnLeft: 90.
      daly go: 100 ].
```

For all these reasons, working with scripts is not easy. In fact, I'm quite convinced that the following two statements reflect your personal experience with scripts.

- Writing long scripts is a painful task.
- Repeating long scripts is boring and error prone.
- When copying complex scripts, the likelihood of making a programming[2] error, such as omitting a line, is high.

Instead, we would like to *define a sequence of messages once and for all;* a sequence of messages, to give the sequence a *name*; and then to be able to *send it as a single message to any robot* — just like the predefined robot messages such as go:, north, jump:...

With this approach we could define a new *method* square, and then write Script 1.3 — but don't execute it yet, because the method square has not yet been defined. But you can see that you do not have to duplicate and adapt the sequence of messages defining a square anymore. You can just use it twice.

I hope you are now convinced that defining the method is worth the effort.

Script 1.3 (*Using the method square)*

```
| pica daly |
pica := Bot new.
daly := Bot new.
daly go: 200.
daly color: Color red.
pica square.
daly square
```

2 How Do We Define a Method?

In this section we give a cookbook recipe for creating a method. In Squeak you can define methods on any object but in this book you will only define methods for robots. Therefore to help you I developed

[2] As opposed to a syntax error, which would be caught quickly by the computer. *Programming* errors, on the contrary, are quite difficult to catch.

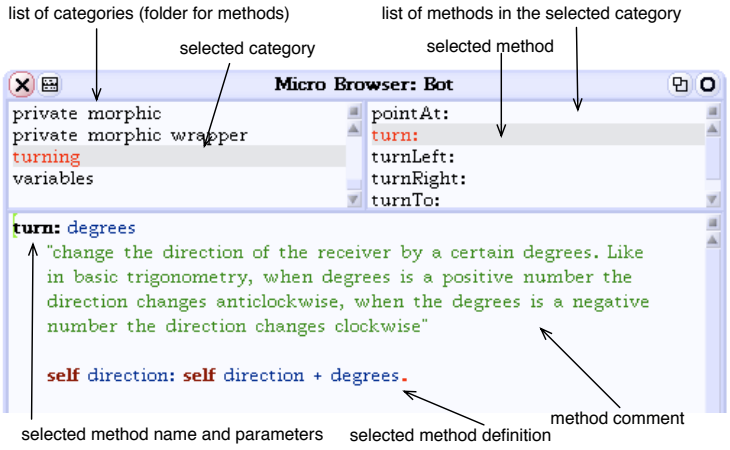


Figure 1.1: A Class Bot Browser showing the definition (in the bottom pane) of the method turn: (selected in the upper right pane) belonging to the category *turning* (selected in the upper left pane).

a specialized code browser named Class Bot Browser just for defining methods for your robots. There is a Class Bot Browser in the working flap, or you can always create one by dragging its thumbnail from the dark blue flap or via the menu **open...**.

Using a Class Bot Browser to define a method requires you to (1) choose or create a method category, that is a kind of method folder (Section 2.2), (2) type the method and (3) then compile it (see Section 2.3).

Let's detail the different parts of a Class Bot Browser.

2.1 A Class Bot Browser

Defining methods requires a new tool: the editor shown in Figure 1.1. This browser is actually a simplified version of the browser used by Smalltalk programmers.

The browser consists of 3 parts or panes:

Categories. The upper left part is the *category list*. It shows the method categories. Method categories are just names that group methods together so that we can find information faster. In Figure 1.1, the category 'turning' is selected; it groups all the operations having to do with robot direction changes. Other categories that group other robot methods are also listed.

Methods. The upper right part is the *method list*. It shows the method names of the methods in the selected category. In Figure 1.1, five methods are listed: pointAt:, turn:, turnLeft:, turnRight:, and turnTo:. The method named turn: is currently selected.

Method Definition. The bottom part is the *code display and code editor*. It shows the definition of the method whose name is selected. This is also where you can type the code of a new method.

2.2 Creating a New Method Category

Methods are grouped by categories. A category is simply defined by a name. To start defining a method, you either define a new category for it, or select an existing category for your method. Let's create a new category named regular polygons.

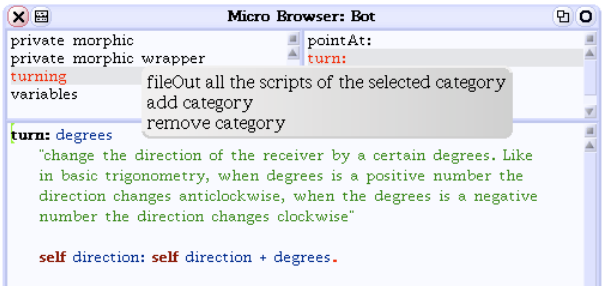


Figure 1.2: To create a method category, open the category menu and select 'add category'.

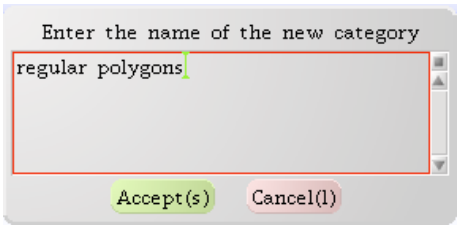


Figure 1.3: Entering the new category name.

1. Click with the right mouse button on the category list. A menu like the one in Figure 1.2 will show up.

2. Select the option add category of that menu.

3. Type the name of the category in the dialog that pops up as shown in Figure 2.2.You may choose any name for the category. Of course, meaningful names are better when you want to share your work with other people or find your method again.

4. Click the Accept button to validate your choice.

As shown in Figure 1.4, the name of the new category appears in the category pane and is automatically selected. The editor is ready to accept the new method definition. It shows you a reminder of how to define a method, which you can remove when you start typing your method.

You are now ready to define your first method.

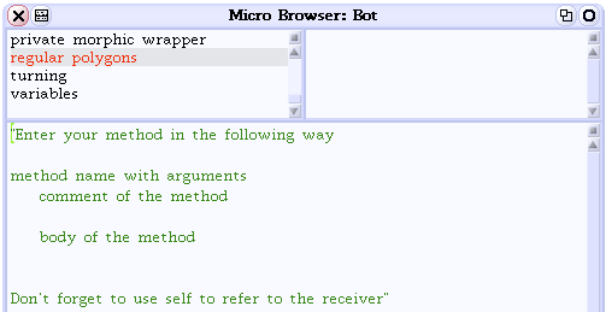


Figure 1.4: The new category is ready.

2.3 Defining your First Method

If the category to which you want to add your method is not selected, select it. Then type the contents of Method 1.1 (shown below) into the code editor pane. To do that, select all the text in the code editor and start typing your method.

Method 1.1

```
square
  "Draw a square 100 pixels wide"

  4 timesRepeat:
        [ self go: 100.
        self turnLeft: 90 ]
```

Defining a method is a three step process:

1. Typing the method. Typing code into the code editor pane works exactly as with the script editor. First delete the reminder text that is in the code editor pane. To make this quick, just point your mouse at the beginning of the editor pane before the first character and click. This will select all of the code editor text. Once you finish typing the new method, your screen should look like Figure 1.5.

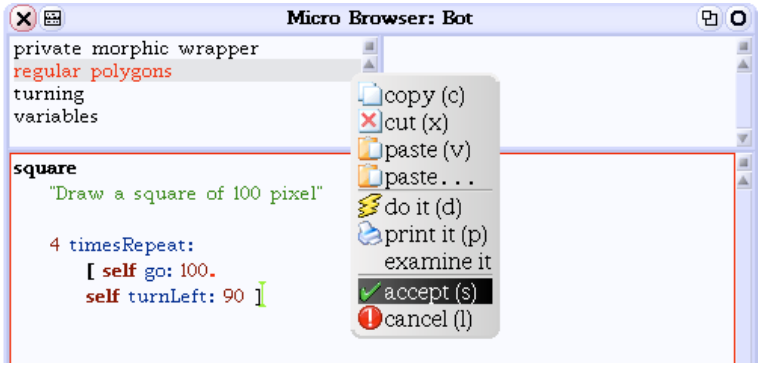


Figure 1.5: You have typed in the method square. Now you should compile it using the code editor menu.

2. Compiling the method. As shown by Figure 1.5, click to bring up the menu for the code editor and select the option **Accept**. This compiles the method that is transforms its definition into a representation that the computer can understand and execute. A new method named square appears in the method list. If you made a mistake while typing the method, Squeak will report the error as it would for a script.

If you define the method correctly, you should be able to compile it without Squeak reporting any errors. The browser will then reflect the fact that the compilation is done and that robots can now understand messages with the new method, by showing the new method's name in the method list (see Figure 1.6).

3. Testing the method! You have not finished yet, because the method you defined could be wrong. You should test it. Execute the Script 1.3. You should get one black and one red square.

Notice that a method can be reused several times, as demonstrated by Script 1.3. This is old news. You have used this fact since the beginning of this book: messages such as go:, turnLeft: and so on, are methods defined in the same way as the method square.

3 What's in a Method?

I asked you to type a method without much explanation. Now is the time to analyze the structure of the method.

A method is composed of a *name*, an optional *method comment* and a *method body* (a sequence of messages) as shown by Figure 1.6. The method name can also contain parameters (see chapter **??**), and the method body can also define local variables using vertical bars | and |.

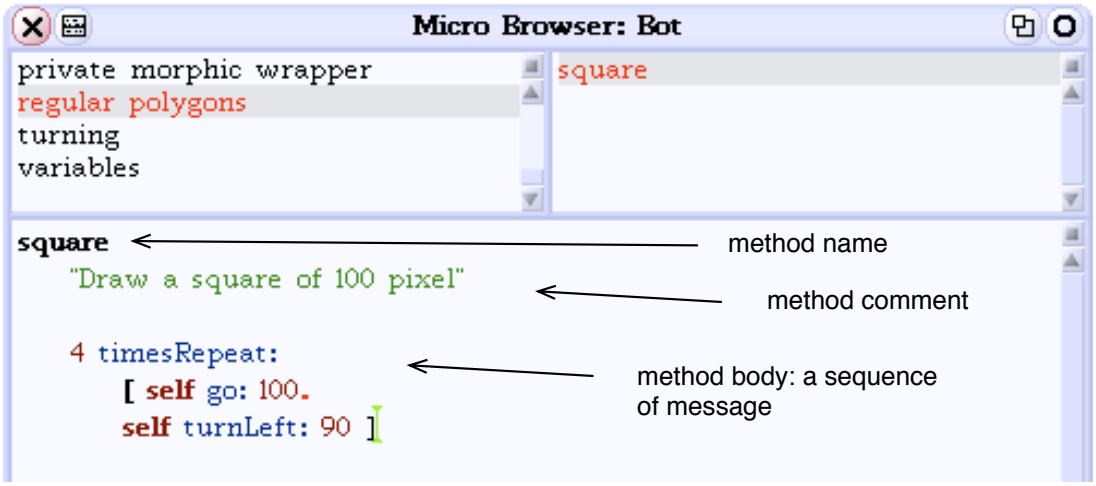


Figure 1.6: A method is composed of a name, a method comment and a method body.

> A method name should always represent what the method does, not how it does it.

Method Name. A method name should always represent what the method does, not how it does it. When you want somebody to open a door, you don't explain all the physics and mathematics involved. It is the same for methods.

Method names without parameters such as square follow the same syntax as variable names. They are composed of alphanumeric characters (letters and digits), and start with a lowercase character. In our case the method name is square.

Method Comment. A comment consists of text enclosed between double quotes ("). The text cannot contain any double quotes. However a comment can be as long as you like, and can continue over several lines.

In general a comment explains the purpose and the effect of the method. It explains how the method can be used, not how the method does its job. Anyone who wants to know how the method works can read the method's body.

If the method name is clear enough, the comment may be omitted. In our case the method comment is: "Draw a square 100 pixels wide"

Method Body. After the comment comes the method definition itself that is the sequence of messages that are executed in response to a message. In our case the method body is:

```
4 timesRepeat:
    [ self go: 100.
    self turnLeft: 90 ]
```

> A method is a named sequence of messages. It is composed of a name, a comment and a sequence of messages. Once a method for robot is defined, any robot can execute it in response to a message with the same name.

3.1 Script vs. Method: an Analysis

Let's compare Method 1.2 with Script 1.4. You can see these differences: (1) The line declaring the variable pica is not in the method; (2) the line creating the robot is not there, either and (3) in the rest of the method, the variable pica is replaced by self.

Script 1.4 (*A simple square)*

```
| pica |
pica := Bot new.
4 timesRepeat:
      [ pica turnLeft: 90.
      pica go: 100 ]
```

Method 1.2

```
square
  "Draw a square 100 pixels wide"

  4 timesRepeat:
        [ self go: 100.
        self turnLeft: 90 ]
```

Remember that a robot method represents a sequence of messages which can be sent to *any* robot: the robot referred to by the variable pica is not necessarily the receiver of the message square. daly could also be the receiver of the message square as we saw in Script 1.4.

Also, the message square is sent to an *existing* robot. There is no need to create a robot since the one we want to send message to already exists. This implies that while defining the method square, you need a way to refer to the object that will receive the message square.

In fact, we want to send the sequence of messages specified within the method body to the specific object receiving the message square and to no other. Therefore we need a way to refer to the receiver of a message. This is the purpose of self. Inside a method, self represents the object receiving the message.

The self variable. If you remember the discussion of Chapter **??**, a variable is just a named placeholder for an object. In particular, I emphasized that the same variable could be used to point to different objects at different times.

In the case of a method, the variable self points to the object that received the message: when the message pica square is executed self refers to the robot named pica, and when the message daly square is executed self refers to the robot named daly.

> Inside a method the variable self represents the object that received the message.
>
> For example: when the message pica square is executed self refers to the robot named pica, and when the message daly square is executed self refers to the robot named daly.

self is a special variable because you cannot change its value. Only Squeak can assign the value of self. That's why self does not have to be declared between vertical bars |. Moreover, self can only the used inside a method definition.

> When the method code needs to send a message to the receiver, send the message to self.
>
> For example, in the method square one need to turn the robot, so the message turn: 90 is sent to self, that is the robot that will receive the message square.

Method or Not: That's the Question. At this stage you may be tempted to go back and convert all the scripts you have written into methods. This is not advisable, because not all scripts are worth turning into a method. In general, one should define a method when the sequence of messages is general enough to be used several times.

4 Returning a Value

A method can also return a value by using the character up arrow ^ also called a *caret*.

Imagine that you want to have a method that returns the distance that a robot should move in one movement. You can define the method maxDistance shown in 1.3. In this example the method is simply returning a numbe, but we could instead return the result of some complex expression.

Method 1.3

```
maxDistance
  "returns the maximum distance to move"

  ^ 100
```

By default every method returns the message receiver. The method 1.4 is equivalent to the method square defined previously. In fact at the end of every method there is an implicit expression ^self. However in this book you do not have to worry about that.

Method 1.4

```
squareEquivalent
  "Draw a square 100 pixels wide"

  4 timesRepeat:
        [ self go: 100.
        self turnLeft: 90].
  ^ self
```

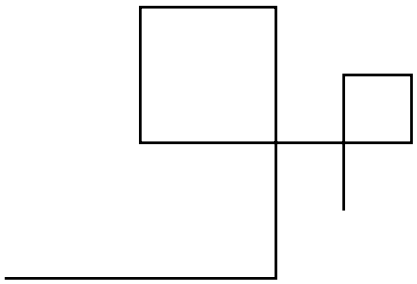
In this book, you will not use this feature much, but it is important to know that a method always returns a value.

5 Pattern Drawing

Now it is time to practice. As you have seen, it is quite easy to transform a script into a method. Many seasoned programmers use scripts to test ideas. When the feasibility of an idea has been proven in the form of a script, they move the code of the script into a method for later reuse. The next exercise trains you to do exactly this. Let's consider the following script 1.5 which draws a geometric shape.

Script 1.5 (*A Simple Pattern)*

```
| pica |
pica := Bot new.
pica go: 100 ;
      turnLeft: 90 ;
      go: 100 ;
      turnLeft: 90 ;
      go: 50 ;
      turnLeft: 90 ;
      go: 50 ;
      turnLeft: 90 ;
      go: 100 ;
      turnLeft: 90 ;
      go: 25 ;
      turnLeft: 90 ;
      go: 25 ;
      turnLeft: 90 ;
      go: 50
```

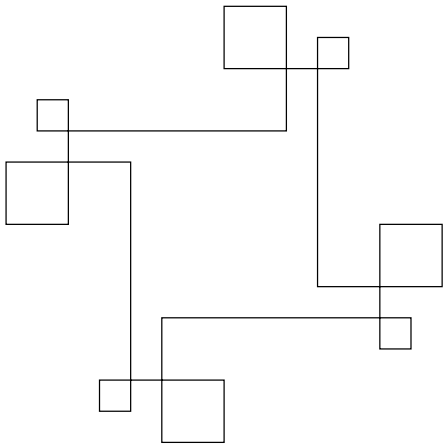


Experiment 1.1
Create a method named pattern which produces the figure drawn by Script 1.5.

You now can use this method in a script to draw a frame.

Script 1.6 (*A Frame)*

```
| pica |
pica := Bot new.
4 timesRepeat: [ pica pattern ; go: 50 ]
```



At this point, some astute reader might ask: why don't we create a method —named frame50, for example— corresponding to that of Script 1.6?. This is indeed possible since *any method created for a robot can be reused by another robot method*. This is the topic of the next chapter.

Experiment 1.2
Create a method named frame50 which produces the picture produced by Script 1.6.

Summary

1. A method is a named sequence of messages. It is composed of a name, a comment and a sequence of messages. Once a method is defined, any robot can execute it in response to a message with the same name.
2. A method name should always represent what the method does, not how it does it.
3. A new method for a robot is created using a class Bot Browser that is a special editor for defining methods.
4. Inside a method the variable self represents the object that received the message.

   When the method code needs to send a message to the receiver, send the message to self.

Glossary

- Method categories. A method category is a folder in which methods are sorted. It helps you find methods faster.
- Method. A method represents a sequence of messages that an object can execute. A method has a name. It is executed when an object receives a message having the same name.
- Browser. A browser is a special tool for viewing and editing methods.
- Comment. A comment is a piece of text surrounded by "" that explains the purpose of a method.
- self. self is a variable predefined by Smalltalk that always represents the receiver of the message in a method definition.