# 1

# A First Script and its Implications

While sending messages using direct interaction with a robot is a fun and powerful way of programming robots, it is a bit of a limited way to write more complex programs. Therefore you shall learn the notion of *script*, *i.e.*, a sequence of expressions and all the necessary concepts that you will use in this book. Hence, this chapter settles down the vocabulary for the rest of the book. It also serves as a map to the next chapters that will introduce in depth the concepts briefly presented in this chapter.

First I present how you to send multiple messages to the same robot separating messages with a semi colon. Then you shall learn how to write script using a dedicated tool called a workspace. I will describe the different elements that compose a script and present the possible mistakes that can happen when writing programs.

## 1 Using a Cascade to Send Multiple Messages

Start to draw a rectangle of 200 height on 100 width. To do so you will start to type the first message go: 100 hit the return key then click on the robot and type the second expression turnLeft: 90 and hit the return key, then click on the robot and type the expression go: 200... As you quickly notice it this is really tedious. What would be much more handy would be to be able to type everything in one shot and that it gets executed.

You can send multiple messages to a robot by separating them with a semi-colon ;. To send to a robot the messages go: 100, turnLeft: 90, and go: 200 simply separate them with a semi-colon ; as follows go: 100 ; turnLeft: 90 ; go: 200 (see Figure 1.1). This way of sending multiple messages to the same robot is called a *cascade* of messages in Squeak jargon.

However, using a cascade (that is sending multiple messages to a robot separated by the character ;) does not work well for complex programs. Indeed even for a drawing a simple rectangle is it too long as shown by the second message in Figure 1.1. In addition, there are other concerns that professional programmers take into account such as if they can store the sequences of messages and replay them later, whether they can reuse them and not have to type them all the time. For all these reasons, we need other ways to program robots. The first way is to write the sequence of messages, that is called a *script*, in a text editor and ask the environment to execute them as you shall see in a minute.

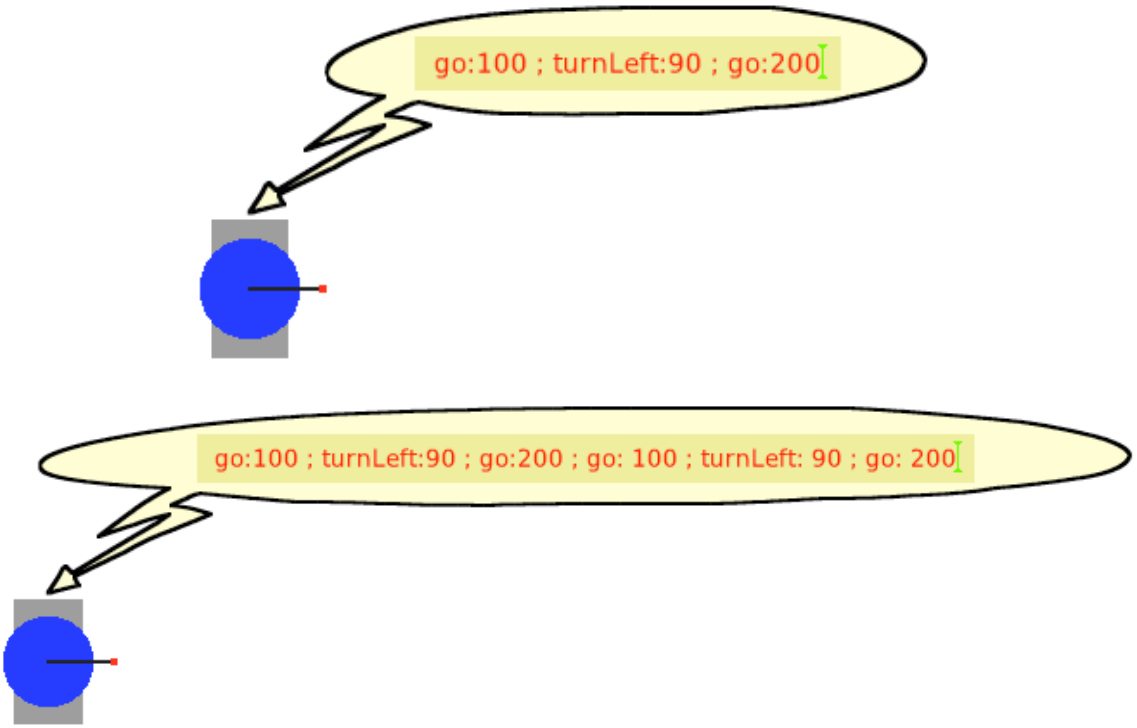


Figure 1.1: Sending several messages in one shot to a robot using ;.

## 2 A First Script

The BotInc environment provides a small text editor, called the Bot Workspace which is dedicated to script execution (that is executing of the expressions composing a script). Click on the bottom flap called **Working**, by default it contains a Bot Workspace editor as shown by Figure 1.2.

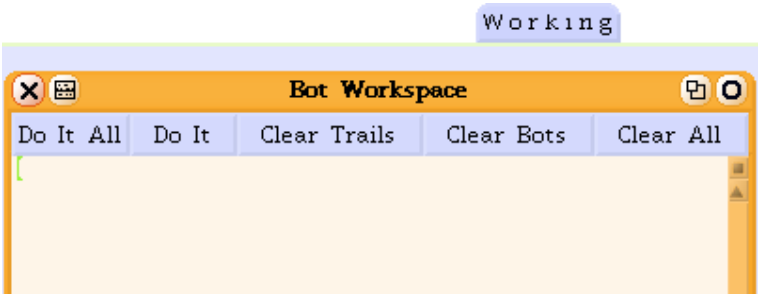


Figure 1.2: A Bot Workspace is a small editor dedicated to the execution of robot scripts.

Let us define a first script that draws a rectangle and explain it in detail (script 1.1). Figure 1.3 shows the script in a Bot Workspace and the result of its execution obtained by pressing the **Do It All** button. Try to get the same result: type the script and press the **Do It All** button. Note that I use pica as a nickname of picasso since our robots are drawing pictures.

**Script 1.1**

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90
```

The **Do It All** button of Bot Workspace executes *all* the messages it contains, therefore before typing a script make sure that no other text is already present in the Bot Workspace. Moreover, computers and programming languages cannot deal with even the most obvious mistakes, so be careful to type the text exactly as it is presented in script 1.1. Pay attention to type the uppercase "B" of Bot on the second line and to end each line with a period. There is no need to put a period at the end of the last line because there is no other messages to send to the robot.

# 3 Squeak and Smalltalk

As you see the script you typed is simple but still it constitutes a program. A program is a list of expressions that a computer can execute. To define programs we need programming languages, *i.e.*, languages that allow programmers to define programs that a computer can understand and execute.

## Programming Languages

The idea behind a programming language is to support programmers to express solutions to their problems. Support means various things like ease of expression of the task, efficient execution of the program, reliability of the resulting application, proof that the program is correct, readibility of the code written, ease to change the application later on ... There is no *per se* best language that satisfies all these properties. Programming languages exhibit different expressive power and properties.

In this book you shall learn how to program in the Smalltalk programming language in the Squeak environment. Smalltalk is a object-oriented programming language like Java or C++.

## Smalltalk and Squeak

Squeak is the name of a programming environment. A programming environment is a set of tools that programmers use to develop applications. Squeak contains a lot of tools: text editors, code browsers, a debugger, an object inspector, a compiler, widgets.... Moreover, Squeak contains a lot more for example, you can program music, animate flash files, access the networks, display 3D objects...and much more! However before you can start programming complex applications you have to learn some basic principles and this is the purpose of this book.

Squeak programmers develop their applications by writing programs using the programming language called Smalltalk. Smalltalk is an *object-oriented* language. This means that objects are created — virtual objects within the computer, of course — and these objects are able to understand messages that they execute when they receive them. However in the context of this book we will not show you

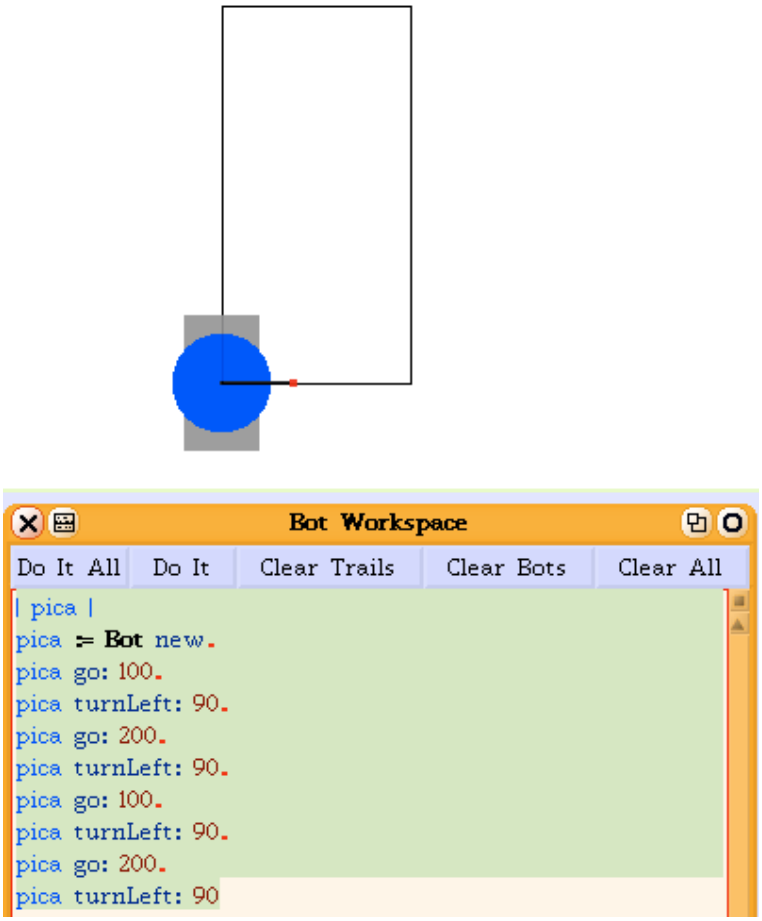


Figure 1.3: A script executed using the **Do It All** button of the Bot Workspace and its result.

how to define new classes, you will only define new behavior for your robot because object-oriented programming is a bit complex and you have to learn basic programming concepts first.

We chose Smalltalk because it is simple, uniform and pure: in Smalltalk everything is an object that send messages to other objects and receive messages. In Smalltalk there are simple rules and they are always applied consistently. In addition, Smalltalk was originally designed for teaching novices how to program. Note that even if Smalltalk was designed for novice programmers in mind, large and complex applications have been written in Smalltalk such as the applications controlling the machines that produce the AMD micro-processors that you may have on your current computer.

Finally, a really interesting aspect of Squeak is that it completely written in Smalltalk. This implies that if you understand Smalltalk you can change, adapt the system to your own needs or learn from the system. With Smalltalk you have a lot of power in your hands. I hope that you get motivated to learn how to program, but watch out programming like playing piano or painting is not simple, therefore do not be discouraged if you get some difficulties, this is normal and I designed this book so that topics get introduced one after the others smoothly.

# 4 Programs, Expressions and Messages

Now we are ready to take a closer look at your first script and explain it.

## 4.1 Typing and Executing Programs

Let us start by looking at the way you wrote your first script or program (script 1.1). You typed a text, a sequence of *expressions*, then you asked Squeak to execute it by pressing the **Do it All** button. Squeak executed the sequence of expressions, that is, it transformed the textual representation of your program into a form that is understandable by a computer, then each expression was performed one after the other. In the first script, the execution created a robot and the robot executed one after the other the messages that were sent to it.

A program consists of a sequence of *expressions* that are executed by the Squeak environment. In this book, we call such a sequence a *script*.

> Vocabulary Point. We called a *script* a sequence of expressions.

A program is a bit like a cooking recipe. A recipe describes all the steps to cook a meal. A program describes all the steps to produce a certain effect. For example to cook pancakes, the expressions are all the steps we should do to prepare them: mix flavor and milk, add the eggs, heat the pan, etc...

## 4.2 Anatomy of a Script

Now comes the time to understand your first drawing script that I copy here (script 1.2):

**Script 1.2**

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90
```

In a nutshell, the script 1.2 declares that it uses a variable named pica to refer to the robot it creates. Once the robot is created and associated with the variable pica the script asks the robot to move to different locations on the screen. Now let us analyze each line step by step even if certain concepts such as variables will be treated in separate chapters.

**| pica |** This first line declares a variable. It tells Squeak that we want to use the name pica to refer to an object. Think of it as saying to a friend, from now on I will use the word pica in my sentences to refer to a robot. You shall learn more on variables in Chapter **??**.

**pica := Bot new.** This line creates a new robot by sending the message new to the robot class named Bot and associates it with the name pica, the variable that we declared before. The word Bot requires an uppercase letter B because it is a class, that is a factory of robots.

**pica go: 100.** The message go: 100 is sent to the robot we named pica. This line can be understood as follows: "pica, move by 100 units of length on the screen". Note that any message name that terminates by a colon indicates that this message needs additional information, such as a length or an angle. For example, go: 100 says that the robot should move 100 pixels. The message name is go:.

**pica turnLeft: 90.** This line asks pica to turn 90 degrees on its left. This lines is again a message sent to the robot named pica.

The other lines are just repetitions of the previous ones, so they have the same effect.

> Any **message** name that terminates by a colon indicates that this message needs additional information, such as a length or an angle.
>
> For example, the message name turnLeft: requires a number representing the angle from which the robot should turn.

**About Pixels.** On a computer screen the unit of distance is called a *pixel*. A pixel is the size of the smallest point which can be drawn on a computer screen. Depending on the type of computer you are using the actual size of a pixel can vary. You can see pixels by looking at the screen through a magnifying glass.

### 4.3 Expressions, Messages and Methods

Now I should clarify some vocabulary points.

**Expression.** We call an expression any element of a program. Here are some examples of expressions:

- | pica | is an expression that declares a variable (see Chapter **??**).
- pica := Bot new is an operation, called assignment, that associates a value to a variable (see Chapter **??**). Here, the newly created robot obtained by sending the message new to the class Bot is associated to the variable pica.
- pica go: 100 is a message send. The message go: 100 is sent to pica.
- 100 + 200 is a message send. The message + 200 is sent to 200.

**Message.** We call a *message*, a pair composed of a *message name* also called a *message selector*, and *message arguments*, the required values to execute the message as shown by Figure 1.4. A message is sent to a *message receiver*.

Example of messages are:

- In pica beInvisible, the message beInvisible is sent to a receiver, a robot. This message does not require arguments.

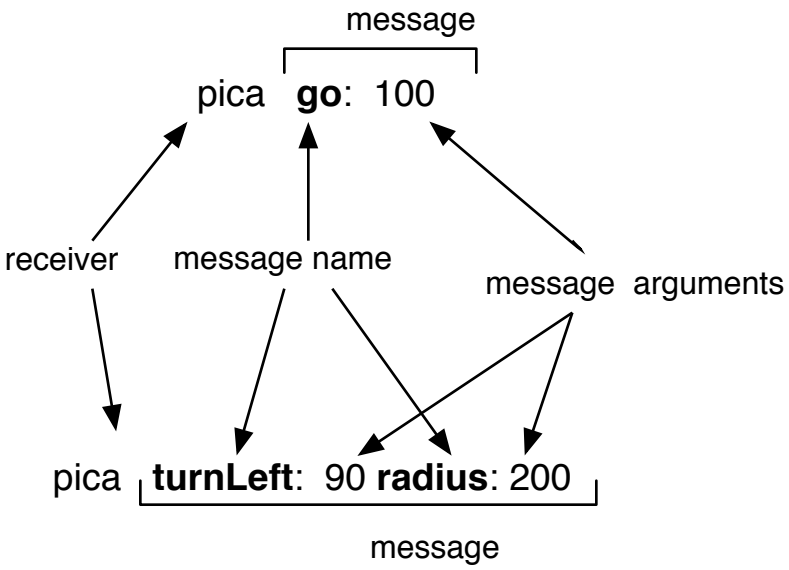


Figure 1.4: Two messages composed of a message name or method selector, and a set of arguments.

- In pica go: 100, the message go: 100 is sent to a receiver a robot. It is composed of the method name go:, and an argument the number 100. Here, 100 represents the length in pixels of which the robot should move. Note that the colon is part of the method selector.

- In 33 between: 30 and: 50, the message between: 30 and: 50 is composed of the method selector between:and:, and two arguments 30 and 50. This method checks whether the receiver, a number, is between two values, here the numbers 30 and 50.

- In 4 timesRepeat: [ pica go: 100 ], the message timesRepeat: [ pica go: 100 ] sent to the number 4, is composed of the method selector timesRepeat:, and the argument [ pica go: 100 ] a block, a sequence of messages (see Chapter **??**).

- In 100 + 200, the message + 200 is composed of the method selector +, and an argument, the number 200. The receiver is the number 100.

**Message separation.** You have noticed that each line of the script 1.1, except the first one, is terminated by a period. As we have said earlier, the first line is not a message. Such a line is called a variable declaration in computer jargon. Thus, we can make the following observation: each message must be separated from the following one by a period. Note that putting a period after the last message is possible but not mandatory: Smalltalk accepts both.

Messages should be separated by a period. The last statement does not require a terminal period.

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

**Hints.** A period. is a message *separator* so there is no need to put one if there is no message after it like at the end of a script or a sequence of messages.

**Method.** When a robot receives a message it executes a *method* which is kind of script that has a name. A method is a named sequence of expressions that a receiver executes in response to a message. For example, the method go: makes the receiver going a certain number of pixels in its current direction. This method go: is executed when a robot receives a message whose name is go:. In the future, I will explain to you how to define new methods for your robot but for now we do not need them to start programming.

## 4.4 Using the Cascade

As I mentioned in the first section of this chapter, you can send multiple messages to the same robot by separating them with a semi-colon, called a cascade. In a script, we can also use a cascade (;) to send multiple messages to the same robot. script 1.2 is equivalent to the following one (script 1.3) where all the messages sent to the same robot are separated by a semi-colon. Using cascades is handy when you want to avoid to type the receiver of the multiple messages. The cascade is interesting because it shortens scripts. However pay attention that you understand clearly that all the messages are sent to the same receiver.

**Script 1.3**

```
| pica |
pica := Bot new.
pica
  go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90 ;
  go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90.
```

To send multiple messages to a Bot use a semi-colon ; to separate the messages, following the template: aBot message1 ; message2.

For example, pica go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90

## 4.5 About Robot Creation

To get a new robot you have to send a message, the message new to the *class* Bot. Note that this is not new! This follows exactly what you did in the previous chapter where you clicked on the blue and orange box named Bot representing the class of the same name and typing new in the bubble. In Squeak, we *always* send messages to objects, robots or classes to interact with them. There is no

difference in treatment, except that class and objects understand different messages. Indeed, an object does not know how to create other objects so sending to a robot the message new leads to an error and sending the message color or go: to a class does not make sense and leads naturally to an error. Still in both cases you are sending messages!

Note that other class may provide other messages to create objects such as the class Color which returns the color blue or green in response to the messages blue or green. Do not worry I will always let you know how to get objects of specific classes.

> To get a new object, send the message new to a class. Bot new creates a new robot. Other classes may offer different messages to get new objects. For example, Color blue asks the class Color to create a new color been blue.

## 5 Possible Errors while Writing Programs

Computers are capable of making highly complex calculations at incredible speed, but they lack the intelligence to correct small mistakes. This means that each expression given to a computer must be given without mistake. The smallest mistakes — even a lowercase letter instead of an uppercase one — are likely to be misunderstood by a computer. In that case, an error message will appear on the screen. This is likely to occur when you are doing your first experiments. So do not despair and try to understand what went wrong.

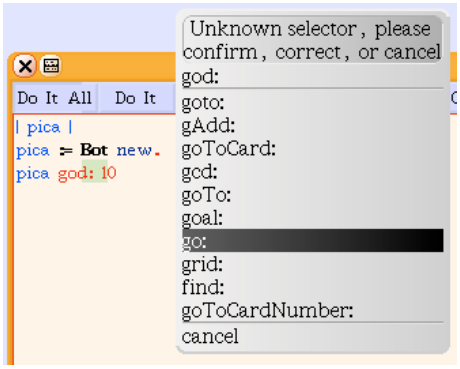


Figure 1.5: We mispell the message go: and type god: instead. The message god: does not exist therefore Squeak prompts us.

Squeak colors the letters while you are typing, when a word is getting red, this means that you are writing something that Squeak does not know. When a word is blue, for a variable or a message, or black for a class, this indicates that everything is correct. In addition, when we execute a message containing an error, Squeak tries to help you by notifying you when it encounters error in the code you wrote. The error messages that Squeak use are in fact menus. The top part in turquoise of the menu window contains a short description of the error; then, depending on the type of error some suggested corrections may be listed as options. Note that you can always cancel the execution by choosing the cancel choice in the menu and in such a case you must find the place that the computer did not understand, correct it, and retry to execute the script. Here are the most common errors.

**Mispelling a message.** You can misspell the name of a message. In Figure 1.5 I misspelt the message go: and typed god: instead. The message god: did not exist therefore Squeak turned the word in red. When I tried to execute the script, Squeak prompted me. Squeak tried to guess what is the message name I wanted to write and it proposed me several messages. I can pick the right message (go:) and the message god: will be replaced by go: or I can simply chose cancel. In that case I will have to change manually god: into go:.

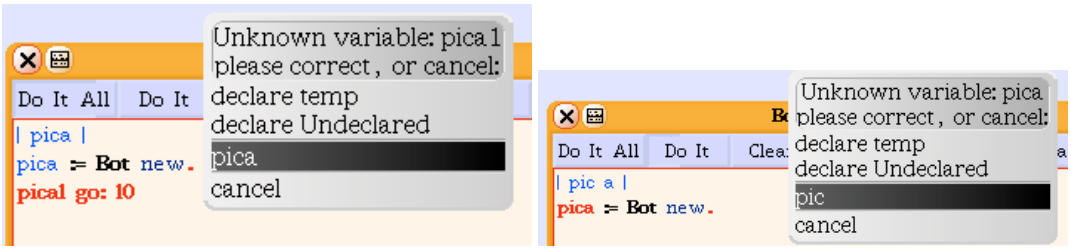


Figure 1.6: Two examples of error. We mistyped the names of the variable. First in the script pica1 has not been declared and second while defining the variable: car o defines two variables pic and a but not a variable pica.

**Mispelling a variable.** There are two ways to mispell a variable during its declaration (between the two vertical bars |) or in the script itself. Figure 1.6 shows the two cases: in the top case I typed pica1 instead of pica in the script. Squeak proposed me to define pica1 as a new variable or to replace pica1 by pica. In the present case choosing pica is the solution as shown by Figure 1.6. The bottom case shows that I misspelt the variable pica during its declaration. In fact the space in the middle has for effect to define two variables pic and a. Squeak saw that pica was not declared and then it proposed me to declare a new variable with this name or to use the variable pic.

**Unused variables.** It may happen that we declare too many variables. Even if this is not an error, *i.e.*, the program can still run correctly. Squeak checks this for us and proposes that we remove the unused variables. For example, in Figure 1.7 the script only uses the variable pica and not pic and a. Therefore Squeak prompts us to remove those.

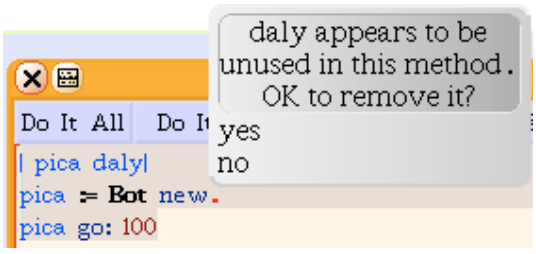


Figure 1.7: All the variables and messages are correct. However the variable daly is defined but not used so Squeak indicates it to us. Unused variables are not a problem so you can proceed.

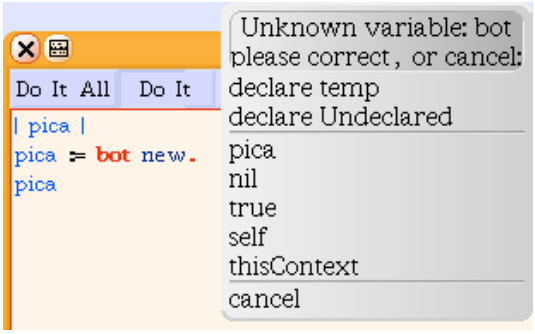


Figure 1.8: We forgot the uppercase B to indicate that we want to send a message to the class (the robot factory). We should correct it.

**Forgetting an uppercase.** Another common mistake is to forget uppercase when needed. Uppercase is needed when you want to send message to object factories. Figure 1.8 shows that I typed bot instead of Bot. Squeak tried to find a solution but it fails. In such a case you have to correct it yourself. In the context of this book you only have to pay attention to put an uppercase for Bot the robot factory, Color the factory of colors.

**Forgetting a period.** Finally one of the most common mistakes that even fluent programmers make, is to forget a period or a cascade between two messages. In such a case Squeak thinks that the following variable is just another message and tries to correct it. For example in Figure 1.9 a period is missing after the expression pica := Bot new therefore Squeak tries to send the message pica and this message does not exist, therefore it tries to propose a replacement but failed to find. In such a case, you have to cancel and type the period manually.

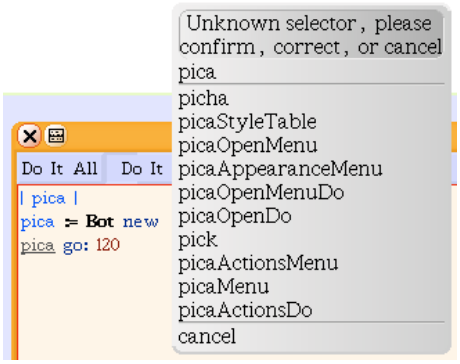


Figure 1.9: Two examples of error. We mistyped the names of the variable. First in the script pica1 has not been declared and second while defining the variable: car o defines two variables car and o but not a variable pica.

**About Colored Words** Squeak tries to identify mistakes when you are typing your scripts and it provides some visual clues on the possible errors. Figure 1.10 shows some typical situations:

- (a) I started to type the first letter of an undeclared or unknown variable. As there is no variable starting with the letter x, Squeak turned the x in red, letting me know that I was doing a mistake.
- (b) I finished to typed a variable which is declared, therefore Squeak turns it into blue to let me know that the variable is declared.
- (c) I'm writing a variable whose beginning is the one of a declared variable so Squeak underline it to let me that I'm going well.
- (d) As soon as I type a character that leads to a variable that is not declared, Squeak turns it in red. Note the difference with the previous case. In case (c) I could have typed the character a and then completed the word pica as in (a), I typed the character b and ended up with a undeclared variable (picb).
- (e) After having typed the name of a declared variable (as in case (b)), I added an extra character a and this leads to the name of an unknown variable (picaa).
- (f) Squeak tries to do the same for the message names. Here I misstyped the message go: and typed instead gou. As soon as I typed the character u, Squeak saw that the message name gou was unknown so it turned it in red.
- (g) Squeak tries to do the same for the classes. Here I typed the character w after Bot and Squeak indicates to me that there is no class named Botw in the system.

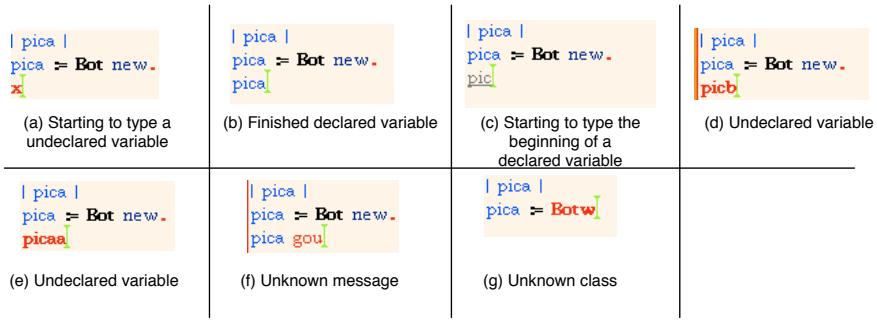


Figure 1.10: Squeak using colors to help finding mistakes.

## Summary

- *To execute an expression*. Press the **Do It All** button of the workspace.
- A *script* is a sequence of expressions that perform a task.
- A message is composed of a message name, and some arguments. Some message may not required arguments. For example pica beInvisible
- Any message name terminated by a colon indicates that this message needs additional information, such as a length or an angle for example. For example, the message name turnLeft: requires a number representing the angle from which the robot should turn.

- To get a new object, send the message new to a class. Bot new creates a new robot. Other classes may offer different messages to get new objects. For example, Color blue asks the class Color to create a new color been blue.
- A class is a factory of objects. Class names always start with an uppercase letter. Here Bot is the factory creating new robots and Color the one creating colors.

  ```
  Bot new color: Color blue
  ```

- Messages should be separated by a period. Terminal period is not required.

  ```
  pica := Bot new.
  pica go: 100.
  pica turnLeft: 90.
  pica go: 100
  ```

- To send multiple messages to the same object use a semi-colon ; to separate the messages, following the template: aBot message1 ; message2. For example, pica go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90 send the message go: 100, turnLeft: 90, go: 200, turnLeft: 90 to the same robot named pica.