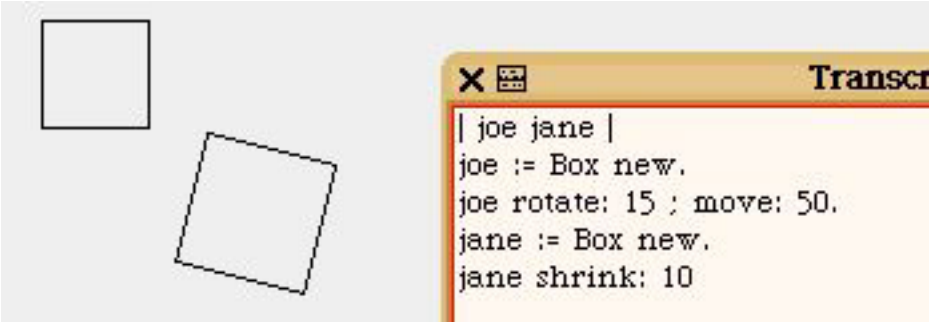
1

Implementing Joe the Box



@@Should we change the class definition template@@

In this chapter you will program your first class, the class `Box`. Doing so you will learn how to describe the behavior and the state of objects. We start with a first implementation that we later refine. This first class is one of the first example used to teach object-oriented concepts to novices by researchers that invented object-oriented programming.

1 Box's Behavior and State

A box has the following behavior, it knows how to draw itself, move to a given distance, move to a given point, rotate, grow and shrink. A typical scenario is described by script **??**. A graphical result is shown by the first figure of the chapter above.

Script 1.1 (*A typical Box scenario)*

```
| joe jane |
joe := Box new.
joe rotate: 15.
joe grow: 100.
joe move: 10@10.
joe moveTo: 150@200.
jane := Box new.
jane move: 30@-30.
jane shrink: 40.
jane rotate: 45
```

It is worth to spend some time looking at script **??**. It shows that while at the first glance the scripts may look similar ot the turtle messages they are not. From an object point of view, there is difference between asking a turtle to draw a square as shown in chapter **??** and asking a box to draw itself. The methods and the arguments that they require are different. Here a box knows how to grow, shrink and rotate.

Before starting programming, we have to analyze the behavior of a box to imagine a possible way to program it. Here is the behavior a box should have. A box should know how to:

- draw itself at a given location. When a new box is created it automatically displays itself.
- move to a given location (method `moveTo: aPoint`).
- rotate from a given angle (method `rotate: anInteger`).
- translate from a certain distance (method `move: aPoint`).

It is fundamental to start by looking at objects from their *behavior*. An object is a behavioral entity, *i.e.*, an entity reacting to messages. A similar behavior can be implemented by different manners so it is crucial not to start to think in terms of the internal structures that may represent the object but in terms of the essence of the object, its *behavior*.

Teacher's Corner

Note the way we phrase the sentences describing box actions: we do not say the box is displayed but it displays itself. We always use the active form where the subject is the object itself. Considering the object as a living being is a good way to think in an object-oriented manner. Imagine talking about an animal or a person you will say that the person acts and not is acted by others.

Teacher's Corner

From behavior to state. Now from this description of the box's behavior, we should imagine a possible state for a box that could be used to implement the wished behavior. As this example and the concept of box are familiar, we propose that boxe state is represented by a size, a position and a tilt.

In fact any box will be represented by such a triplet (size, position and tilt) but each given object will have its own triplet values. For example, the box referenced by the variable `joe` in script **??** has its *own* state, *i.e.*, its own size, position, and tilt. In the same way the box `jane` has a *similar* state because it is also a box created from the class `Box` too but it has its own state which may or not equal to the one of `joe`. When the state of one given box changes it does not change the state of the other boxes. This situation is illustrated by the first figure of this chapter. If this aspect is not clear we suggest you to (a) read chapter **??** and (b) create the class `Box`, inspect two instances and modify their states.

2 Defining the class `Box`

To create a class we use a dedicated browser called the system browser or class browser as presented in chapter **??**. To open such a browser, bring up the default menu and chose the menu item **open...** and the item **browser** (or use the b).

To create a class, create first a new category (which represents a folder for all the classes we will create related to this small project) by selecting the item **add Item** of the menu associated with the leftmost pane of the browser (as explained in chapter **??**). Name it for example `JoeTheBox`. When you select the newly created category, the system displays a template to help you defining a new class (see class **??** and the figure **??**).

Class 1.1

```
Object subclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'JoeTheBox'
```

Modify the proposed template to obtain class **??** and in the bottom pane bring the menu and choose the menu item **accept**. Now the class exists. The system shows you that the class is defined by displaying it

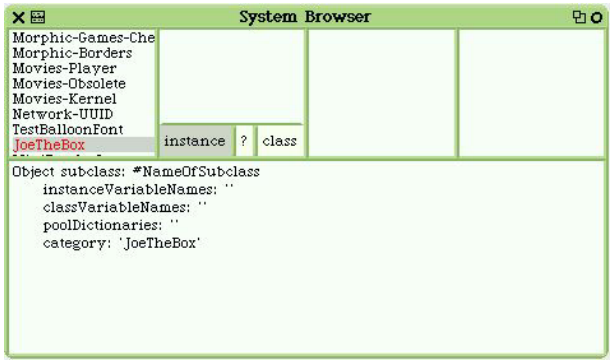


Figure 1.1: The browser shows you that a new class has been created by displaying it in the second pane from the left.

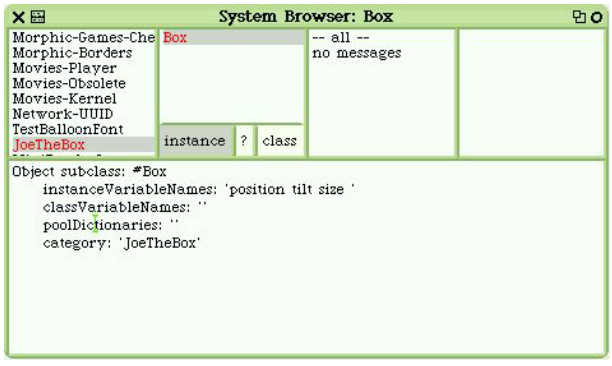


Figure 1.2: The browser shows you that a new class has been created by displaying it in the second pane from the left.

in the second pane as shown in figure **??**. Using the terminology used in other programming languages we can say that the class has been *compiled*. This means that we could already create instances of this class, even if now this is not really useful since they do not have any specific behavior.

Class 1.2

```
Object subclass: #Box
   instanceVariableNames: 'position size tilt '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Joe The Box'
```

Here are some explanations about the class definition **??**: A box is a simple object. Hence, it is a subclass `Object` (The concept of subclass and inheritance is explained chapter **??**). The internal state of box instances, such as the boxes `joe` and `jane`, is represented by instance variables of the class `Box`. So line 2 we specify that the class `Box` has three instance variables by given their respective names. Here the class `Box` has the instance variables `position`, `size`, and `tilt`. This indicates to the class `Box` to create instances having three values representing the box's state. As shown by class **??** we empty the other parts of the templates because they are irrelevant for now.

Important!

A class acts as an object factory, an instance model, or a mould. The instance variables describe the state that the instances created by the classes will have. Each instance of the class will have the structure described by the class but filled with *its own* values.

Instances have their own state but it follows the description given by the class. Two instances of the same class can have different values for the same instance variable but they cannot have a different number of instance variables.

Teacher's Corner

The *factory* or mould metaphor is really useful to explain the difference between classes and instances. Here, the class `Box` describes and creates boxes of different size, position and tilt but all have these three characteristics.

Teacher's Corner

3 Initializing Instances

Once the class is defined, create and inspect one of its instances by executing script **??** or by using an inspector (see chapter **??**). The figure **??** shows an inspector on a `Box` instance. All the instance variables have `nil` as value. Indeed, when an instance is created by invoking the method `new` on a class, the *default* behavior of the class is to return an *uninitialized* instance. Uninitialized means that all the instance variables of the newly created instances have no value. To represent the *no value* concept, Smalltalk uses the special object `nil` [1]. That's why the instance variables of the inspected box have all as value `nil`.

Script 1.2 (*Inspecting a box)*

```
Box new inspect
```

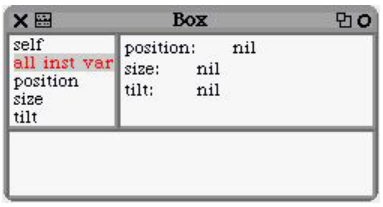


Figure 1.3: Inspecting a *non initialized box*: all its instance variables are empty, *i.e.*, having `nil` as value.

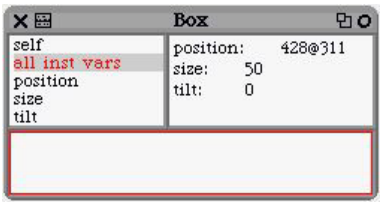


Figure 1.4: Inspecting an *initialized box*: all its instance variables contains some values coherent with their role.

Having uninitialized values is not really good because methods may not work or have to test if the variables have been correctly initialized. But even then this is not satisfactory because if an instance variable is not initialized it is difficult to know the value to initialize it. In fact the best solution is to initialize the instance as soon as it is created.

For that purpose we specialize the method `initialize` that sets up a default state for a box. The method `Box»initialize` is automatically invoked by the method `new` on newly created instances. This method sets the instance variables values. Once this method defined, in the bottom pane of the inspector evaluate the expression `self initialize`. If you closed the inspector or want to convince you that the method `initialize` is invoked when a new instance is created, reuse script **??** to check that the created instance is now well initialized. In both cases, you should obtain a situation similar to the one described by the figure **??**.

Method 1.1

```
Box>>initialize
   "A box is initialized to be in the center of the screen, with
   50 pixels size and 0 tilt"

   size := 50.
   tilt := 0.
   position := World bounds center
```

Reader Note

The fact that the `new` method automatically call the `initialize` method is a little extension we added into Squeak. It may happen that such an extension will be included into Squeak at the time you will read this book. In any case the plain Squeak solution to this problem is explained in chapter **??** so that you can understand and program in Squeak without our little extension.

Reader Note End

[1]Nil comes from the latin nihil which means nothing.

4 Accessing Instance Variables

The method `initialize` above illustrates an important aspect of the object model of Squeak. The instance variables are accessible from the methods as if they were defined in the method body. For example, we are assigning `50` in the instance variable `size`. The instance variable `size` is accessible from any method of the class `Box`.

Contrary to the of a script (`| caro | for example`) which do not exist after the script execution, instance variables last the complete object life-time. We propose you some experiments to really understand this phenomena below. Note that this behavior is not new, we used it constantly with the turtle. For example, we changed the direction of the turtle using the method `north` which was somehow changing the internal turtle state representing its direction, then later used the direction to perform some other actions.

We propose you to do some experiments to really understand this notion. Define the methods `size` (method **??**) which returns the value of the instance variable `size` and `size: anInteger` (method **??**) which changes the value of the instance variable `size` to be the one specified as argument. Such a kind of methods are called *accessor* methods because they only get and set information in instance variables. The code `^size` returns the value of the instance variable size.

> In Squeak to return a value, use the character ^ followed by the expression whose value has to be returned.

Method 1.2

```
Box>>size

   ^size
```

Method 1.3

```
Box>>size: anInteger

   size := anInteger
```

Now execute script **??** and use the menu item **print it** to get the results we present in script **??** using *returns*. If you have an inspector opened on a box instance, you can also execute the messages `self size`, `self size: 10` in the bottom left part of the inspector. Perform some other experiments to prove yourself that you understand.

Script 1.3 (*Instance variables life-time)*

```
| joe jane |
joe := Box new.
joe size. returns 50
joe size: 10.
joe size. returns 10
joe size: 20.
joe size. returns 20
joe size: joe size + 5.
joe size. returns 25
jane := Box new.
jane size. returns 50
```

Now that you understand better instance variables, you should notice that instance variables are private information of objects so creating *accessor* methods like the methods`size` and `size:` above that only access and return the value of an instance variable open the privacy of instances. We say that the break the encapsulation of the object data. We will discuss in chapter **??** the use of accessors methods.

In summary we have:

> Instance variables are accessible to all the methods of a class. Instance variables last the same life-time than the object to which they belong to.

> In Squeak, instance variables cannot be accessed from outside of an object. Instance variables are only accessible from the methods of the class that define them.

5 Drawing a Box and Other Operations

Now that we initialize a newly created box using the method `initialize`, we are in position to define methods without been worried about the initialization of instance variables.

During your experiments you may want to clear the screen. Use the script **??** for that purpose.

Script 1.4 (*Clearing the screen)*

```
World clearTurtleTrails
```

Method `draw`. We define the method `draw` that uses a turtle but we hide it as shown by method **??**. We create a method, put it at the right position of the box, set the direction of the turtle to the tilt of the box, use the black color and then draw a square of the box size.

Method 1.4

```
Box>>draw
   "Draw the receiver position in black"
   "Box new initialize draw"

   | aTurtle |
   aTurtle := Turtle new hidden.
   aTurtle jumpAt: position.
   aTurtle turnRight: tilt.
   aTurtle penColor: Color black.
   4 timesRepeat: [aTurtle go: size.
                   aTurtle turnLeft: 90]
```

As soon as the method is defined and all the methods it calls exist, it is possible to invoke it. Test the method by executing the code `self draw` into the bottom pane of an inspector in a similar way than shown in figure **??**, or by executing script **??**.

Script 1.5 (*sending the message draw to a box)*

```
| joe jane |
joe := Box new.
joe draw
```

Experiment 1.1

Up until now, creating a new box did not displayed it. Change the method `initialize` so that any new box is automatically displayed.

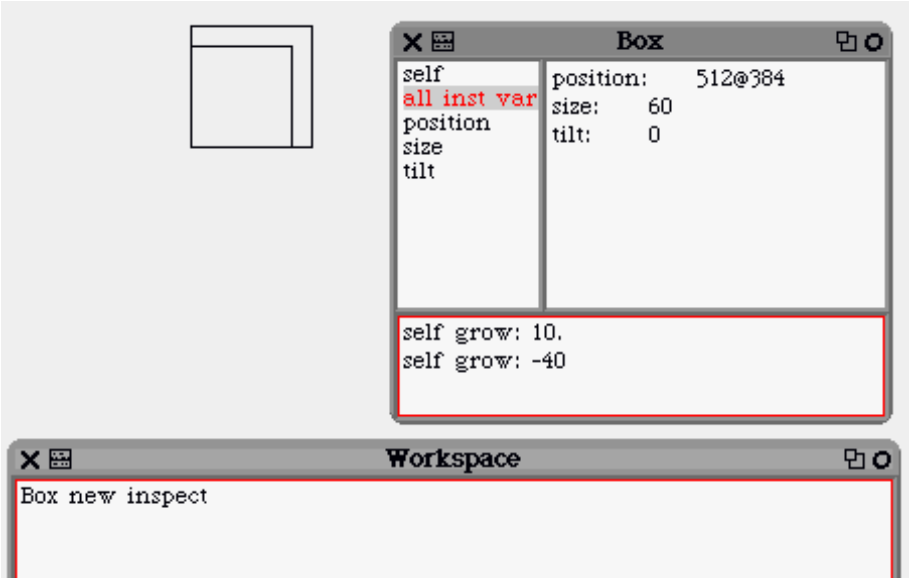


Figure 1.5: A problem with the `grow:` method. The box is not erased when it changes its size.

Method grow: The method `grow: anInteger` makes the box growing of a certain size and redraw itself to reflect this size change. Propose a definition and use the inspector or dedicated scripts to tests your method.

Method 1.5

```
Box>>grow: anInteger
   "grow the receiver's size from increment"

    size := size + anInteger.
    self draw
```

We propose the definition **??** for the method `grow:`. However, try script **??** to see that we have a problem and propose a solution.

Script 1.6 (*Problem with the first `grow:` method.*)

```
| joe |
joe := Box new.
joe grow: 20.

joe grow: 40
```

The problem we have is that the turtle grows and redisplay itself well, but it does not remove the previous box shape. To solve that problem we propose you to define a method named `undraw` which is similar to the draw method except that it draw the box using a transparent color (method **??**).

Method 1.6

```
Box>>undraw
   "erase the receiver"

   | aTurtle |
   aTurtle := Turtle new hidden.
   aTurtle jumpAt: position.
   aTurtle turnRight: tilt.
   aTurtle penColor: Color transparent.
   4 timesRepeat: [aTurtle go: size.
                   aTurtle turnLeft: 90]
```

Now that the method `undraw` is defined, the method `grow:` should call it before anything else as shown by method **??**.

Method 1.7

```
Box>>grow: increment
   "grow the receiver's size from increment"

   self undraw.
   size := size + increment.
   self draw
```

To be able to execute the script we presented at the beginning of the chapter, implement the methods

- `move: aPoint` which translate the box from a distance in x and y specified as a point.
- `moveAt: aPoint` which move the box to the specified point.
- `rotate: anInteger` which rotates the box of a given angle.
- `grow*: anInteger` and `shrink*: anInteger` that make grow and shrink the receiver by a given factor.

6 Limiting Duplication

The methods `draw` (method **??**) and `undraw` (method **??**) are nearly the same except for the color of the turtle. This is not really good, since every times we will change one method we will have to change the other and there is chance that we forgot. Note that on this really simple example this is not really a problem but we want to show you a good principle.

Propose a solution to this problem. The idea is that to avoid duplication, the methods `draw` and `undraw` can call a third method with the color of the pen as argument. We could named this method `drawWithColor:`. Try to implement such a method before reading the solution.

The `drawWithColor: aColor` (method **??**) shows a possible implementation, it factors out the duplicated code. Now change the methods `draw` and `undraw` to call this method with the right argument.

Method 1.8

```
Box>>drawWithColor: aColor
   "Draw the receiver using a given color"

   | aTurtle |
   aTurtle := Turtle new hidden.
   aTurtle jumpAt: position.
   aTurtle turnRight: tilt.
   aTurtle penColor: aColor.
   4 timesRepeat: [aTurtle go: size.
                   aTurtle turnLeft: 90]
```

In general we should avoid as much as possible to have duplicated code. This is not a problem to duplicate code for a small experiment. However, if you want to keep the code always think that you should create other method to share and reuse the duplicated code. Creating one or several methods to factor the duplicated code is a good trick to cure duplicated code.

> Avoid duplicated code. Refactor the duplicated code by calling a method representing the duplicated code.

7 Looking at Alternate Designs

We said that the implementation we proposed is one of the multiple ways of implementing the behavior of the `Box` class. In this section we want to look at other possible implementations. First let us analyze the current implementation. The class `Box` has its own state via the instance variables `tilt`, `position`, and `size` then gives a part of its state to a turtle, its tilt and position. The `Box` class uses the `Turtle` class to realize its behavior. This is a common practice where a class do not repeat behavior but reuse the behavior of an existing class.

We used the class `Turtle` because it was familiar to us. However, another class, the class `Pen` could have been a possible candidate too. Look at the class `Pen` and change the method `drawWithColor:` to use it instead of `Turtle`. What is important is that the interface proposed by the class `Box` should not changed. We are changing the internal implementation of the class `Box` but this should not change its behavior.

Now if we look carefully we see that the turtle or the pen instance are created every time the box is draw and undraw. In addition the state of the box is systematically copied to the turtle state then lost because a turtle is recreated and the previous one is lost. One idea would be to use a turtle as representing part of the box state. Indeed a turtle has also a position and a tilt.

Define the class `BoxT` as shown below in class **??** and reimplement some of the box's methods to convince you that this is possible. This solution has the advantages that less objects are created, less state is copied from the box to the turtle and as drawbacks that the class `Box` is tied with the class `Turtle`.

Class 1.3

```
Object subclass: #BoxT
   instanceVariableNames: 'size turtle'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Joe The Box'
```

To help you we show two methods, method **??** and method **??**, that are important. Try to do it by yourself first. Implement all the other methods.

Method 1.9

```
BoxT>>initialize
   "A box is initialized to be in the center of the screen, with
   50 pixels size and 0 tilt"

   size := 50.
   turtle := Turtle new hidden.
   turtle jumpAt: World bounds center.
```

Method 1.10

```
BoxT>>drawWithColor: aColor
   "Draw the receiver using a given color"

   turtle penColor: aColor.
   4 timesRepeat: [turtle go: size.
                   turtle turnLeft: 90]
```

8 What you should have learned

Blabla here...