[**1**] []**1** []

1 Programming a Simple Morph

In this chapter you will apply what you learned in the previous chapter. This is a good repetition before implementing the Bot environment. For that purpose we propose you to build your own turtle. This way you will check how to create a simple Morph, and how you can define a class by refining another one and reusing its properties.

For the sake of simplicity we will not really recreate exactly the same turtle than the one defined in the class `Turtle`. However, we will show the important points that have to be addressed when defining a new class by specializing an existing class. We will show you how to obtain a simple turtle class only with a couple of methods. While doing this exercise we will show you how to take advantage of the interactivity of Squeak.

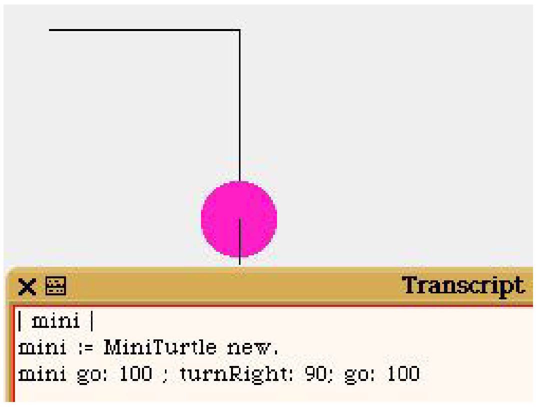
You should be confortable with the practices presented in this chapter because they are basis of object-oriented programming and they will be used repeatedly during the project Bot the Robot proposed in the chapter **??** and its subsequent chapters. In this chapter we will first show you how to define a class then we will analyze all the steps we took to do so and stress the important points that you should understand.

1 The MiniTurtle class

The mini turtle like its big cousin, the normal turtle, is a *graphical* entity living in the world of Squeak. To represent such an entity we use the graphical library of Squeak called Morphic. The easiest way to create a graphical class in Squeak is to define a new class which inherits from the class `Morph` or one of its subclasses and defines the desired behavior. For more complex graphical objects, one needs to do the same and in addition combine several other graphical objects. For the mini turtle we only need the first approach.

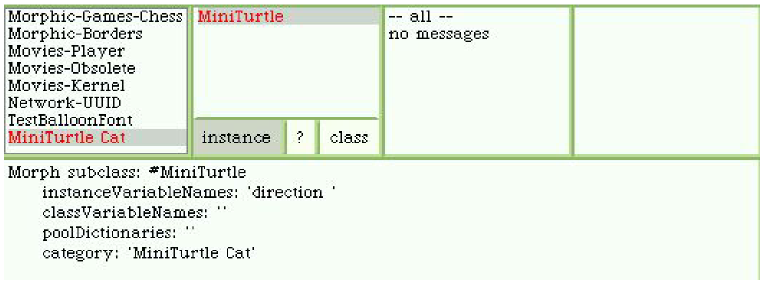
Script 1.1 (***A mini turtle in action @@Picture to change@@***)

```
| mcaro |
mcaro := MiniTurtle newStandAlone
           openInWorld.
mcaro jump: 100 ; go: 100 ;
      turn: -90 ; go: 100
```



```
| mini |
mini := MiniTurtle new.
mini go: 100 ; turnRight: 90; go: 100
```

To create graphical objects, they have to be instances of graphical classes,i.e., classes that inherit directly or indirectly from the class `Morph`. The class `Morph` represents the functionality that any graphical



```
Morph subclass: #MiniTurtle
    instanceVariableNames: 'direction '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MiniTurtle Cat'
```

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

object can have.[1]. Subclasses of `Morph` refine such functionality in various ways as the impressive inheritance hierarchy shows it (in a browser select the class Morph and ask an hierarchy browser h).

A mini turtle has the same functionality than a normal one as shown by the script script **??**.

To create a class, create first a new category (which represents a folder for all the classes we create related to this small project) by selecting the item **addItem of** the menu associated with the leftmost pane of the browser (See Chapter chapter **??**). Name it for example `MiniTurtle Cat`. Select the newly created category, the browser shows you an empty class template that you should fill. Clearly the class `MiniTurtle` should be a subclass of the class `Morph` now we shoudl think about the instance variables we need. Note that you can compile using the **accept** menu item the class and then latter add instance variables.

To implement the behavior of a mini turtle we need to be able to represent its location on the screen and its direction. The class `Morph` already provides the first one via the methods `position` and `position:` because any morph should be able to know its position on the screen.

Fill this template to obtain the definition presented in class **??** and select compile it using the menu item **accept**. This means that the class `MiniTurtle` inherits from the class `Morph` (this is equivalent to say that the class `Morph` defines a new subclass as shown by the template. Remark that this second form is used by the system because the class `MiniTurtle` does not exist while Morph does, so we can send the message `subclass:...` to the class `Morph` and no message to the class `MiniTurtle` as it does not exist yet). Then the class definition states that the class `MiniTurtle` has one extra instance variable named `direction` than its superclass, the class `Morph`.

Class 1.1

```
Morph subclass: #MiniTurtle
   instanceVariableNames: 'direction'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MiniTurtle Cat'
```

Note that as soon as you compiler a class using the menu item **accept** the class exists now. The system shows you this information by displaying the newly created class in the second pane from the left as shown in the figure **??**. 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 extra behavior or state than a normal morph.

Now we will define methods one by one and show how as soon as the method is defined it can be used. Execute the following expression, you should obtain a blue box in the right corner of your screen as shown by the figure **??**.

[1]The Morphic system is powerful, but is from a software engineering point of view in bad shape. It may happen that at the time you will read this book, the system already get refactored. In any case we use a limited set of functionality offered by the Morphic system.

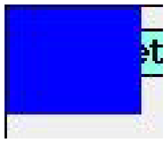
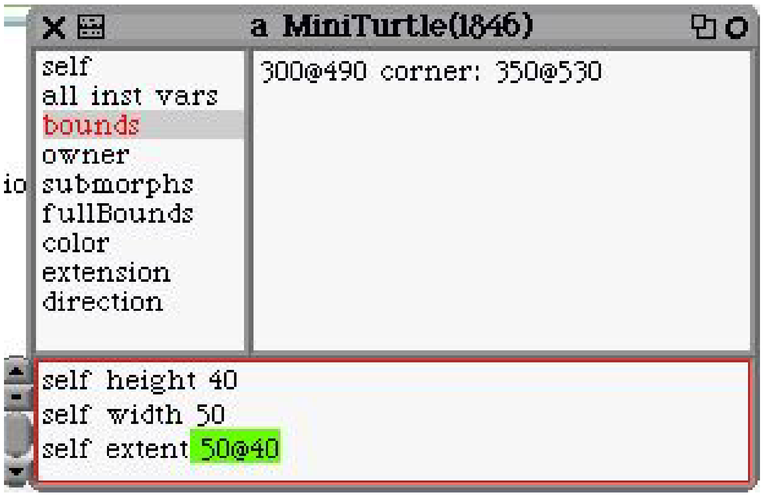


Figure 1.2: The class `MiniTurtle` does not specify how its instances are drawn so the default way defined by the class `Morph` is used. We obtain a blue rectangle whose size represents the extent of the morph.



```
300@490 corner: 350@530
```

```
self height 40
self width 50
self extent 50@40
```

Figure 1.3: Using an inspector to query the state of object. Here the method `height`, `width`, `bounds` and `extent` have been invoked.

Script 1.2

```
MiniTurtle newStandAlone openInWorld
```

The class `MiniTurtle` does not specify yet how its instances are drawn so the default way defined by the class `Morph` is used. The surface of the morph is filled by the color blue. This default behavior is generic and not really interesting even if lot of the morph behavior is already possible such as changing its color, size....

As shown in figure **??**, use an inspector (see chapter **??**) to get the extent of the box. You should noticed that the extent is a rectangle. As we would like to draw a circle for rendering the turtle we would like that the extent be a square. We will fix this in the following.

2 Initializing a Mini Turtle

To change the extent of newly created mini turtle, we can send the message `extent:` as follow, `MiniTurtle newStandAlone openInWorld extent: 40@40`. However, this approach is tedious and more important, if we pass our code to another person, such a person may not know that he should to that. In fact this is the responsibility of the mini turtle to initialize its state well. For that purpose we will specialize the method `initializeToStandAlone` which is called by the method `newStandAlone` defined on the class `Morph`.

Method 1.1

```
MiniTurtle>>initializeToStandAlone

   super initializeToStandAlone.
   self extent: 50 @ 50.
```

Define the method `initializeToStandAlone` in the class `MiniTurtle` . Now each time a mini turtle is created by the method `newStandAlone` the method `initializeToStandAlone` is invoked and the newly created mini turtle is are well initialized. Create a new mini turtle and convince you that its extent has been well initialized.

Completely Initializing `MiniTurtle`. The direction of the mini turtle is not initialized nor its color. So redefine the `initializeToStandAlone` method to take into account the initialization of the direction and the color of the mini turtle which should be pink when created.

Method 1.2

```
MiniTurtle>>initializeToStandAlone

   super initializeToStandAlone.
   self extent: 50 @ 50.
   direction := 0.
   self color: (Color r: 1.0 g: 0.097 b: 0.774)
```

It is the responsibility of a class to initialize correctly its instances, not the responsibilities of its users. Use for that the method `initialize` that is invoked by the creation method `new`. For Morphs use the method `initializeToStandAlone` that is invoked by the method `newStandAlone`.

3 Drawing a Mini Turtle

Now we create a graphical representation for the mini turtle. The key point when drawing a morph is that its graphical representation should not draw outside of the morph extent, else when the morph is moved it will let dirt on the screen.

Drawing a morph is done by applying graphical operations such as drawing ovals, rectangles or lines into a canvas. The system performs some complex operations to display a morph and we only have to specify the `drawOn:` method whose responsibility is to draw the morph graphical aspect.

The method `Rectangle»fillOval:color:` that draws an oval of a given size and of a given color requires as second argument a rectangle that represents the shape of the oval drawn. The script **??** shows how we obtain a rectangle (here with size of the same length) of `50` width centered around the point `100@100`.

Script 1.3

```
   Rectangle center: 100@100 extent: 50
```

In the class `MiniTurtle` , defines the following method, accept it, then click on the blue box. It should now appear like a pink circle. Note that the circle does not take the complete place of the morph extent because we want to have some space left for drawing the direction in which the turtle is pointing at.

Figure 1.4: Computing a point at a certain distance following a given direction from a starting point

Figure 1.5: Difference between the position and center of a morph.

Method 1.3

```
MiniTurtle>>drawOn: aCanvas

   aCanvas
      fillOval: (Rectangle center: self center
                           extent: self height - 10)
      color: self color.
```

What you see is that as soon as the new method has been defined and compiled by the system, it is possible to use it. As the method we just defined is defined on the class `MiniTurtle` and that the instance (displayed previously as a blue box) is from this class, the system now automatically calls this method instead of the one defined on the class `Morph`.

Now to indicate the direction to which the turtle is pointing at we draw a line from the center of the morph towards that direction.

The expression `self center + (direction degreeCos @ direction degreeSin negated * headLength)` in the method below computes the point that at the distance `headLength` from the center of the turtle following the direction to which the turtle points as shown by the figure **??**.

Note that there is a difference between the position of a morph and its center as shown by the figure **??**. The position is the top left corner of a morph.

Method 1.4

```
MiniTurtle>>drawOn: aCanvas

   | headLength |
   headLength := self height // 2 - 2.
   aCanvas
      fillOval: (Rectangle center: self center
                           extent: self height -10)
      color: self color.
   aCanvas
      line: self center
      to: self center
            + (direction degreeCos @ direction degreeSin negated
                 * headLength)
      width: 1
      color: Color black
```

In this method we see that the result of the method `center` is used at three different places, so we can compute it once and reuse the result as shown in the final version of this method.

Method 1.5

```
MiniTurtle>>drawOn: aCanvas

   | headLength center |
   center := self center.
   headLength := self height // 2 - 2.
   aCanvas
      fillOval: (Rectangle center: center extent: self height -10)
      color: self color.
   aCanvas
      line: center
      to: center
            + (direction degreeCos @ direction degreeSin negated
                 * headLength)
      width: 1
      color: Color black
```

Now from within an inspector, change the value of the direction instance variable and click on the mini turtle. The line indicating the direction will point in this direction. In fact you only see the effect when you will click on morph because at this time the morph is redraw.

4 Some more operations

Now we define some of the operations that a mini turtle can perform. For that we need to define a method to make turtle draw line on the screen. We give the definition of such a method method **??** but we will not explain this method. Just define it and compile it on the class `PasteUpMorph` (menu item **find class** in the leftmost pane).

Method 1.6

```
drawLineFrom: oldPoint to: newPoint color: aColor size: anInteger
   "Draw a line from a point to a new point with aColor and a size"
   | origin offset |
   oldPoint = newPoint ifTrue: [^ self].
   self createOrResizeTrailsForm.
   origin := self topLeft.
   turtlePen sourceForm width ~= anInteger
ifTrue: [turtlePen squareNib: anInteger].
   offset := anInteger // 2 @ (anInteger // 2).
   turtlePen color: aColor.
   turtlePen
drawFrom: (oldPoint - origin - offset) asIntegerPoint
to: (newPoint - origin - offset) asIntegerPoint.
    self invalidRect: ((oldPoint rect: newPoint) expandBy: anInteger)
```

Now we are ready to define the behavior of the mini turtles. Let us start with the `jumpAt:` method that positions a mini turtle at a given position represented by a point. As we consider that the position of a turtle is its center, here we just need to change the center of the morph that represents the mini turtle.

Method 1.7

```
MiniTurtle>>jumpAt: aPoint
   "change the receiver's position so that its center is placed
   at the position aPoint"

   self center: aPoint
```

Now this is easy to define the method `goAt:` as shown in the method definition method **??**. It just draws a line from the current center to the final point and it just there. The expression `trailMorph` returns the screen in which the turtle evolves.

Method 1.8

```
MiniTurtle>>goAt: aPoint
   "make the receiver go at a given point of the screen. The receiver
   lets a trace on the screen from its current position to the final
   point."

   self trailMorph
      drawLineFrom: self center to: aPoint color: Color black size: 1.
   self center: aPoint
```

The method `go: distance` moves forward a mini turtle, just have to invokes the method `goAt: aPoint` by specifying a point that is in the direction to which the receiver points at and from the given distance (See method definition method **??**).

Method 1.9

```
MiniTurtle>>go: distance

   self goAt: self center
             + (direction degreeCos @ direction degreeSin negated
                  * distance) asIntegerPoint
```

Similarly the method `jump:` does the same but by invoking `jumpAt:` instead of `goAt:` (see method definition method **??**).

Method 1.10

```
MiniTurtle>>jump: distance

  self jumpAt: self center
             + (direction degreeCos @ direction degreeSin negated
                  * distance) asIntegerPoint
```

However, there is something not really satisfying in the methods method **??** and method **??**, we define twice the same complex expression. Worse this expression was already used in the method `drawOn:` (See method definition method **??**). Such an expression is based on the fact that the direction of the turtle changes according to the mathematical sense (anticlockwise), now if we want to change it and make it clockwise we will have to fix these three methods. May be while doing that we will forget one of the methods and then the code will be broken. So the right approach is to create a method called for example `positionInDirectionForDistance:` and to invoke from all the places.

So let's do it, we start by defining the method `positionInDirectionForDistance:` (method **??**) then we invoke from the other methods (methods method **??**, method **??**, and method **??**. Note in particular that the resulting methods are shorter and much more readable.

Method 1.11

```
MiniTurtle>>positionInDirectionForDistance: aDistance
  "return the point that is at a distance aDistance in the direction
  pointed by the receiver"

  ^ self center
      + (direction degreeCos @ direction degreeSin negated
            * aDistance) asIntegerPoint
```

Method 1.12

```
MiniTurtle>>go: distance

   self goAt: (self positionInDirectionForDistance: distance)
```

Method 1.13

```
MiniTurtle>>jump: distance

  self jumpAt: (self positionInDirectionForDistance: distance)
```

Method 1.14

```
MiniTurtle>>drawOn: aCanvas

   | headLength center |
   center := self center.
   headLength := self height // 2 - 2.
   aCanvas
      fillOval: (Rectangle center: center extent: self height -10)
      color: self color.
   aCanvas
      line: center
      to: (self positionInDirectionForDistance: headLength)
      width: 1
      color: Color black
```

Note that the process is not wrong, we started to define our methods then we realize that some code could be shared and we refactor it to share the new definition. There is no problem not to see upfront that the definition of the position computation could have been shared. This is mistake not to refactor the code.

Teacher's Corner

You may wonder why we did not proposed directly the good solution with the code refactored but this is on purpose that we made it that way to show the process. It would have been also really good to show that if we change the convention for the turtle direction we would have to fix the code in three different places. We did not make it to avoid slowing down the chapter.

Teacher's Corner

Finally we have to define the method `turn:`, `turnLeft:`, `tunrRight:` as follows:

Method 1.15

```
MiniTurtle>>turn: degrees

   direction := direction + degrees

MiniTurtle>>turnLeft: degrees

   self turn: degrees

MiniTurtle>>turnRight: degrees

   self turn: degrees negated
```

5 Fondamental Aspects of Object-Oriented Programming

Now we would like to stress some key points of object-oriented programming by analyzing the implementation of the mini turtle.

The method `initialize` that we shown in the method **??** and that is repeated hereafter in the method **??** illustrates a key point of object-oriented programming. We can invoke methods defined on the superclass, here `Morph`, as if they were defined in the class `MiniTurtle` . Here the methods `extent:` and `color:` are defined in the class `Morph` but we invoke them using the pseudo-variable `self` like any other methods defined in the class `MiniTurtle` .

Method 1.16

```
MiniTurtle>>initializeToStandAlone

   super initializeToStandAlone.
   self extent: 50 @ 50.
   direction := 0.
   self color: (Color r: 1.0 g: 0.097 b: 0.774)
```

The case of the method `initialize` is different because we are re-defining this method, which is originally defined on the class `Morph`, on the class `MiniTurtle` **and** we want that the method (`Morph»initialize`) defined on the class `Morph` to be still executed. In that case, using `self` to invoke the method `Morph»initialize` would lead to a loop, the method calling itself without any condition to stop this infernal circle. That's why another pseudo-variable, named `super` representing the the receiver of the message exists. `super` allows one to invoke methods overriden, *i.e.,* method defined in superclass and redifined in subclasses.

Note that if a method exists in a superclass and a subclass redefines it without using `super` to invoke it, this ancestor method is just not called.

As a rule of the thumb, always use `self` to invoke method except when you are defining a method whose already exist in the superclass and that you want to invoke. Never use `super` with a different method name that the method which contains it.

Method 1.17

```
MyClass>>badPractice

   ... super anotherMethodName
```

Always use `self` to invoke methods, even if there are defined in superclass of your class.

Only use `super` to invoke methods when you are defining a method that is already defined in a superclass and that you want this method to be still executed.

Here are the guidelines that you should follow when you use accessor method for your own class or for classes defined in the system.

Be Consistent. Apply systematically direct access or accessor methods and not to mix both inside a class. This is extremely confusing for the reader.

Be Private. If you use accessors you should always consider that accessors are kind of private methods that should not often be called from outside of a class. When you call an accessor from outside of the class this should raise a bell and you should check if what you are doing is correct.

Respect your parents. Avoid to access directly superclass instance variables. Even if it is possible to directly access instance variables of a superclass, it is a bad practice. It links outrageously the implementation of the superclass with the one of the class.

Do not directly access instance variables of superclass.

6 Experimentations

It would be nice to be able to be able to change:

- the color of the turtle trace by sending the message `inkColor:`. Introduce such a functionality, and
- the size of the pencil use by sending `pencilSize:` as shown by the following script.

Script 1.4

```
| mini |
mini := MiniTurtle newStandAlone openInWorld.
mini inkColor: Color orange ; pencilSize: 3.
```