1

Coordinates, Points and Absolute Moves

Up to now the messages sent to a robot to move it were relative to its current position. Indeed, the expression pica go: 100 means that you ask a robot to go forward 100 pixels from its current position in its current direction. Such a move is said to be *relative* because the position reached by the robot at the end of the move depends on its initial position. This kind of move is very powerful but sometimes you would like to be able to say to a robot to move to a specific location on the screen such as the middle of the screen, that is make a robot move in a *absolute* manner. For this purpose we need a coordinate system, that is a way to represent a specific location on the screen. You use a coordinate system when you are looking for a street on a map. The map listing indicates that a given street is located in a square identified by one letter and a number or two letters or two numbers. The same applies to computer screen. You can refer to a location given a point that is a pair of numbers. Therefore in this chapter I start to present points and coordinates, then I present new robot behavior and some experiments. This will help us to explore new problems in the future such as using robots to simulate animal behavior.

1 Points

Since everything in Smalltalk is an object, locations on the screen are *also* described by objects called *points*. Points are created by the class Point and their behavior is close to the mathematical one. In two dimensions a point is composed of two coordinates: x (the point horizontal coordinates) and y (the point vertical coordinates). A point is created by sending the message @ sent to a number. For example, the point D of Figure 1.1 is created by the expression 45@35. A's x is 45 and y is 35.

> 200@400 is a point whose x is 200 and y 400.

The following script 1.1 presents how to access point constituents.

Script 1.1 (*Point element access)*

```
| point1 |
point1 := 45@35.
point1 x
-> 45
point1 y
-> 35
```

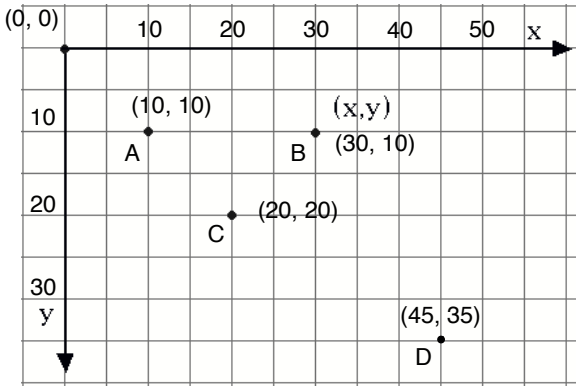


Figure 1.1: A point represents a location in a two dimension area. A point has an x or horizontal coordinates and a y or vertical coordinates.

It is worth to pay attention that the coordinate system in Smalltalk is not completely the same that the mathematical one. Figure 1.1 shows that contrary to the mathematical model, the y axis goes positive from the top to the bottom of the screen while the x axis goes increasing from left to right. We say that in Squeak the origin of the coordinate system is the top left corner of the screen. In a mathematical system the y axis goes from the bottom to the top. Thus, the point `45@35` is located 45 pixels from the left side and 35 pixels from the top of the screen.

> The Smalltalk coordinate system has its origin (0,0) on the top left corner of the screen and y axis goes positive from top to bottom.

All kinds of mathematical operations are available on the points. In this chapter we will only present some of operations that we shall use in the future. Smalltalk goes a step further that proposing the basic mathematical operation. For example, we can multiply a point by a value to get a point whose values is the previous ones but multiplied by this value `(100@200) * 3`, or use common mathematical operation such as addition, substraction on points themselves. Note that the binary operations such as -, *, + creates new points. The script 1.2 shows different operations.

Script 1.2 (*Point manipulation)*

```
| point1 point2 point3 |
point1 := 200@400.
point2 := point1 * 2
point2
—Printing the returned value:    400@800
point2 x
—Printing the returned value:    400
point2 y
—Printing the returned value:    800
point3 := (50@60) + point1.
point3 x
—Printing the returned value:    250
point3 y
—Printing the returned value:    460
point1 + 200      "200 is considered as the point 200@200"
—Printing the returned value:    400@400
```

2 Using Grids

To help you to understand points, you can have a look at the information displayed in the balloon that pops up when you let the mouse over your robots. You can also use a grid. Indeed, Squeak can draw a grid on the background of the screen. To get the grids, bring the **world** menu, select **playfield options...**, and the menu items **use standard texture** or **make graph paper...**. Choosing the last one lets you define the size and the color of the grid.

As shown by script 1.3, you can also program the grid using the following methods: drawGrids and undrawGrids, to draw and undraw the grids, gridColor: aColor to change the color of the grids, gridSize: anInteger to specify the size of the grid, gridWorldColor: aColor to change the color of the world when the grid is drawn, and worldColor: aColor to change the color of the world. You can also get the size of the grid using the method gridSize.

Script 1.3 (*Setting the grids.)*

```
| env |
env := BotEnvironment default.
env gridSize: 25.
env gridWorldColor: Color paleBlue.
env gridColor: Color blue.
env drawGrids
```

Using the **World** menu you can change the size of the screen. To go on full screen, bring the world menu, select **appearance...** and then the full screen on or off menu item. You can also use the following methods fullScreenOff fullScreenOn as shown in the script 1.4.

Script 1.4 (*Setting the screen size)*

```
BotEnvironment default fullScreenOn
```

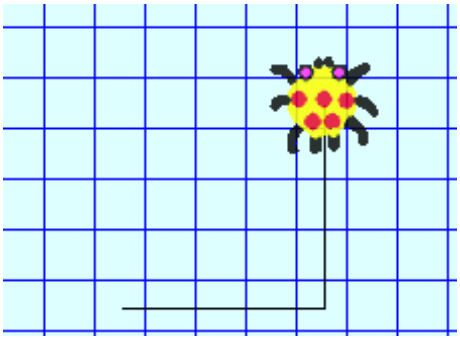


Figure 1.2: Grids of 25 pixels.

3 Source of Errors with Points

The way points are created may lead to some errors as shown by the first line of the script 1.5. 50@60 + 200@400 returns aB3dVector instead of a point representing the sum of the two points. The problem is that when a point, and not an integer, receives the message @, it returns another kind of point that does not interest us. Therefore we should pay attention to the way the messages are sent as explained below.

Script 1.5 (*Possible error with points)*

```
50@60 + 200@400
->  a B3DVector3(250.0 260.0 400.0)
"returns a 3D vector but not a point"

(50@60) + (200@400)
-> 250@460
```

> To avoid trouble with points, surround them with parenthesis when they are involved in complex operation.

As explained in chapter **??**, you should know how messages are executed and in particular

- that parentheses () are evaluated first,
- that *unary* messages are executed before *binary* ones and that *binary* messages are executed before *keywords-based* ones and
- that messages of the same kind are evaluated from left to right.

Here the method @ is just a binary method like any other and it has the same priority that binary methods such as +, *, or //. Therefore the expression 50@60 + 200@400 is executed as it would have been typed as follows (((50@60) + 200) @ 400). The message @ will be sent to a point and not an integer. Let us look at what happens in the first line of the script 1.5 which does not correctly returns the point 250@460 as we expected. The script 1.6 shows how the messages are executed.

Script 1.6 (*Decomposing 50@60 + 200@400)*

```
50@60 + 200@400  is equivalent to (((50@60) + 200) @ 400)

 Step 1
   @ is sent to 50 with the argument 60 , it returns the point 50@60.
 Step 2
   + is sent to the point 50@60 with the argument 200 , it returns
   250@260 because when a number is passed as an argument it is
   considered as the point having the same value for x and y.
   Here 200@200.
 Step 3
   @ is sent to 250@260 with 400 as argument, the object
   B3DVector3(250.0 260.0 400.0) is returned.
```

Now when we simply put parenthesis around the points we obtain a point that is the sum of the two others as explained by the script 1.7.

Script 1.7 (*Decomposing (50@60) + (200@400))*

```
(50@60) + (200@400)

 Step 1
   Parenthesis are evaluated first.
 Step 1.1
   @ is sent to 50 with 60 as argument and returns a point.
 Step 1.2
   @ is sent to 200 with 400 as argument and returns a point.
 Step 2
   + is sent to 50@60 with argument 200@400 and
    returns the new point 250@460.
```

In summary we suggest you to put parenthesis around points when you are doing point manipulation.

4 Absolute Moves

Now that we can specify a location on the screen, we can ask a robot to go directly to a given location. For this task two methods goTo: aPoint and jumpTo: aPoint exist.

- Sending the message goTo: aPoint to a robot, asks it to go to the location represented by the point.
- Sending the message jumpTo: aPoint to a robot asks it to jump to the location represented by the point.

Note that the message jump: and jumpTo: do not leave a trace, while go: and goTo: do. Let us practice now. Try to guess what script 1.8 does. As another experience, try to guess the size of your screen by positioning a robot as close as possible to the bottom right corner.

Script 1.8 (*Going directly at a location and jumping.)*

```
| pica  |
pica := Bot new.
pica goTo: 200@400.
pica jumpTo: 300@400.
pica go: 1.
pica jumpTo: 400@400.
pica goTo: 450@400
```

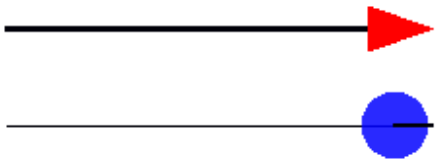
The following section will stress the difference the methods go: aDistance and goTo: aPoint, and jump: aDistance and jumpTo: aPoint.

5 Relative vs. Absolute Motions

Now we shall look at the difference between the methods go: aDistance and goTo: aPoint. The method go: asks a robot to move of a given distance *along its current direction*. Thus, the robot position depends on its current location and of its current direction. Script 1.8 illustrates this.

Script 1.9 (*Parallel motion)*

```
| pica marge |
pica := Bot new.
marge := Bot new.
marge lookLikeTriangle.
pica lookLikeCircle.
marge color: Color red.
marge penSize: 3.
marge north.
marge jump: 50.
marge east.
pica go: 200.
marge go: 200.
```

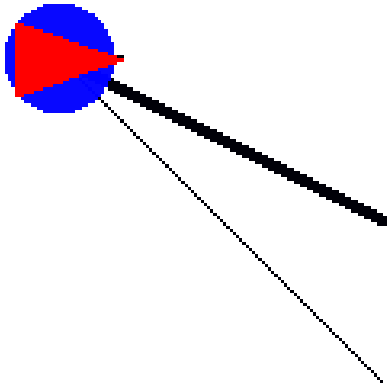


As you can see, the two robots are moving along parallel lines and do not end up in the same location, even though the same message go: 200 was issued to them at the end of the script.

On the contrary, the method goTo: aPoint asks a robot to place itself to a fixed location *regardless* of its position and direction before the move. This is illustrated by script 1.10.

Script 1.10 (*Convergent motion)*

```
| pica marge |
pica := Bot new.
marge := Bot new.
marge lookLikeTriangle.
pica lookLikeCircle.
marge color: Color red.
marge penSize: 3.
marge north.
marge jump: 50.
marge east.
pica goTo: World center - 100.
marge goTo: World center - 100.
```

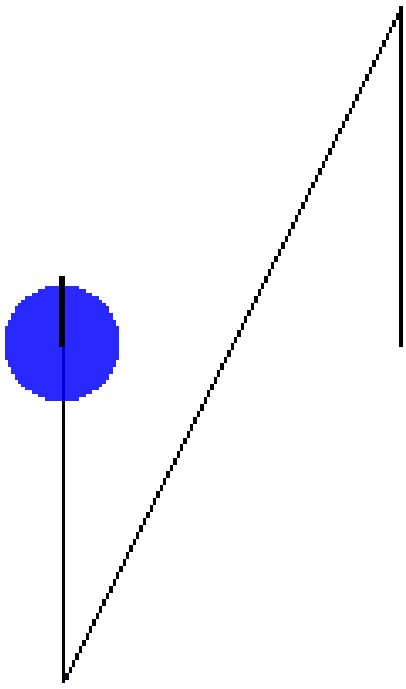


In this case, the two robots end up at the same location. One says that the method go: produces a *relative* motion, whereas the method goTo: produces an *absolute* motion. In the previous script we use the expression World center - 100 so that you get exactly the same picture as the one we show even if you computer has a different resolution than the one used to write this book.

Finally note that the methods go: and goTo: do not change the direction of the robot. This is illustrated by script 1.11. In this script we ask a robot to move forward 100 pixels from its current position, then asks it to go directly to a position that is located at 100, -100 from the center of the screen, and then move forward again 100.

Script 1.11 (*Combining absolute and relative motions)*

```
| pica |
pica := Bot new.
pica lookLikeCircle.
pica north.
pica go: 100.
pica goTo: (World center - (100@-100)).
pica go: 100.
```



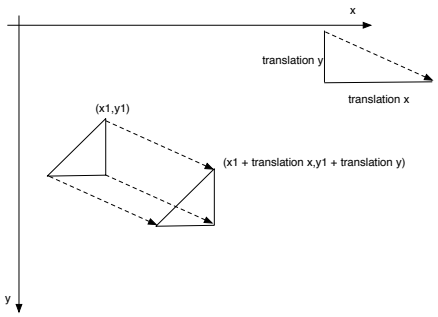


Figure 1.3: Translating a shape requires to add to each point of the shape the same amount for each axis.

6 Some Experiments

Now we propose you some experiences you get more familiar with the concepts presented. As you can see, the robot does not change its direction when it is transported to a given location using goTo:

Experiment 1.1
Using the methods goTo: and jumpTo: define a method rectangleTopLeft: point1 bottomRight: point2 that draws a rectangle. For example, pica rectangleTopLeft: 200@500 bottomRight: 350@700.

Experiment 1.2
Define another method rectangleOrigin: point1 extent: point2 that the second point does not represent anymore the opposite corner but the size of the rectangle. For example pica rectangleOrigin: 200@600 extent: 350@500 we will have a rectangle starting at location 200@600 and having a height of 350 and width of 500.

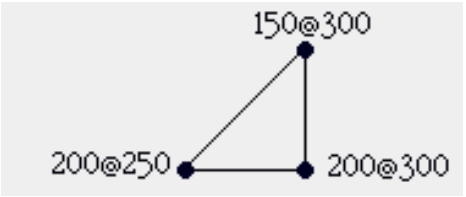
7 Translations

When we add the same amount to all the points of a shape, we obtain the same shape but in another position. This operation is called a translation in mathematics. As shown in Figure 1.3 the amount that we add to the points of the original shape can be different for the x and y axis, therefore it can be represented as a point and the translation is simply expressed as the sum between a vertice and a point representing the translation. A vertice of the new figure is equal to the sum of a point of the translated figure and the translation point. In Figure 1.3 the vertice (x1, y1) is translated by adding the point translation and the result is a point whose $xnew = x1 + translation\ x$, and $ynew = y1 + translation\ y$. For example, the point 200@300 translated by the translation point 50@75 is 250@375.

Triangle. Define a method named triangleAt: point1 point2: point2 point3: point3 which draws a triangle between the points given as arguments. The script 1.12 illustrates how to use such a method.

Script 1.12 (*Using triangleAt:point2:point3:*)

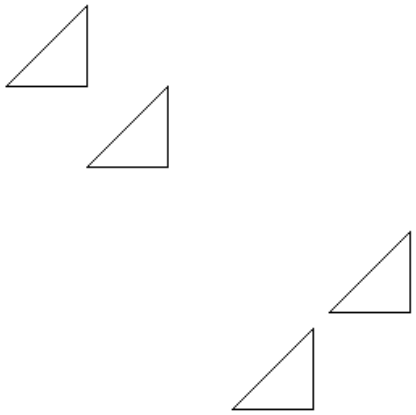
```
| pica |
pica := Bot new.
pica
  triangleAt: 200@300
  point2: 200@250
  point3: 150@300
```



Translating Triangles Now we can draw several triangles by simply translating the first one. In script 1.13, we define three translations and draw the corresponding triangles.

Script 1.13 (*Using triangleAt:point2:point3:*)

```
| pica  point1 point2 point3 t1 t2 t3 |
point1 := 200@300.
point2 := 200@250.
point3 := 150@300.
t1 := -50@-50.
t2 := 90@150.
t3 := 150@90.
pica := Bot new.
pica beInvisible.
pica triangleAt: point1 point2: point2 point3: point3.
pica triangleAt: point1 + t1 point2: point2 + t1
      point3: point3 + t1.
pica triangleAt: point1 + t2 point2: point2 + t2
      point3: point3 + t2.
pica triangleAt: point1 + t3 point2: point2 + t3
      point3: point3 + t3.
```

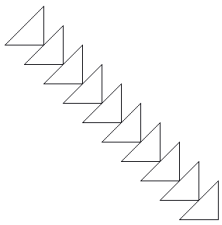


We could also have define another method triangleAt:point2:point3:translation: that performs the translation instead of having to do ourselves the point addition. Such a solution is safer since we cannot apply different translations to different points of the same triangle and therefore we suggest you to implement it.

Flying Geese. One can repeat the translation operation to obtain repeating patterns. The script script 1.14 generates the pattern called *flying geese* in patchwork. Note that we chose the amount of translation so that the next triangle is in diagonal.

Script 1.14 (*Flying Geese)*

```
| pica translation point1 point2 point3 |
point1 := 200@300.
point2 := 200@250.
point3 := 150@300.
translation := 25@25.
pica := Bot new.
10 timesRepeat:
        [ pica trian-
gleAt: point1 point2: point2 point3: point3.
    point1 := point1 + translation.
    point2 := point2 + translation.
    point3 := point3 + translation].
```



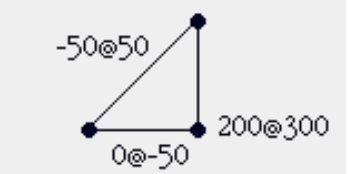
Script 1.15 shows how we can write the translation in a more concise way using the fact that we can multiply points too.

Script 1.15 (*Flying Geese)*

```
| pica  times |
pica := Bot new.
times := 1.
10 timesRepeat:
    [ pica triangleAt: 200@300
            point2: 200@250
            point3: 150@300 translation: (25@25) * times.
    times := times + 1].
```

Experiment 1.3

As a variation of the same problem, define another method named triangleAt: aPoint delta1: aPoint1 delta2: aPoint2 that starts to draw a triangle at the point aPoint then uses the two following arguments as differences between points by reference to the first point. So that t triangleAt: 200@300 delta1: 0@-50 delta2: -50@50 draws the same triangle as: t triangleAt: 200@300 point2: 200@250 point3: 150@300



8 Absolute Moves at Work

Now you may wonder why does one need points? So far, all drawings did not require points. In fact executing most drawings using points would have been quite difficult. Imagine how difficult can be to draw a pentagon only using the goTo: message. Still absolute positions are useful. The following illustrates such an example. We will use a point to keep track of the robot position at a given moment

during the execution of a complex drawing. Then we will used this position to continue our drawing from this place.

The first example is based on the script **??** of chapter **??** in which we were drawing the letter A. At the end of Section **??** we noted that the bottom half of the left bar of the "A" is drawn twice by the script **??**. We considered that it was not that big a problem. Our solution is to use a point to store the location of the robot, location to which the drawing requires to come back later.

Let us modify the script **??** as shown in the script 1.16 to get the possibility to draw an A letter in a relative way.

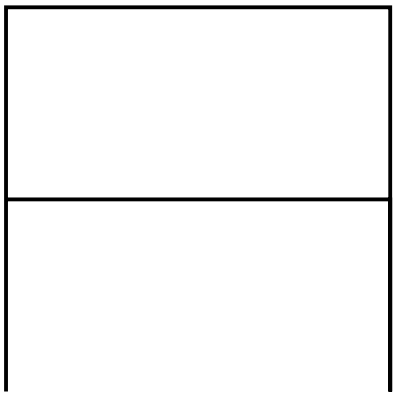
Script 1.16 (*An A relative)*

```
| pica |
pica := Bot new.
pica turnLeft: 90.
pica go: 100.
pica turnRight: 90.
pica go: 100.
pica turnRight: 90.
pica go: 100.
pica turnRight: 180.
pica go: 50.
pica turnLeft: 90.
pica go: 100
```

Now we modify this script as shown in the script 1.17.

Script 1.17 (*The letter "A" is any direction)*

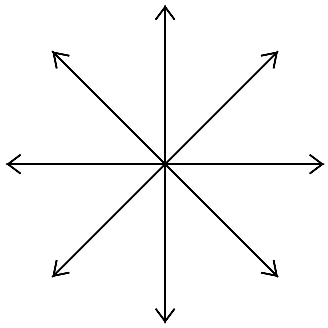
```
| pica barPoint |
pica := Bot new.
pica turnLeft: 90.
pica go: 40.
barPoint := pica center.
pica go: 60.
pica turnRight: 90.
pica go: 100.
pica turnRight: 90.
pica go: 100.
pica jumpTo: barPoint.
pica turnLeft: 90.
pica go: 100
```



In script 1.17, the first vertical bar of the "A" is drawn in two steps first 40 then 60 pixels. Between the steps, the absolute location of the robot is obtained using the method center . This is the location where the bar of the "A" should be drawn. The result of that method is stored in the variable barPoint. After the last vertical bar is drawn, the robot goes back to the location of the bar using the method goTo: barPoint. From this point on, the bar is drawn.

By the way you can verify that the letter "A" is indeed drawn correctly in any direction by adding a command turnLeft: or turnRight: after the creation of the robot.

Experiment 1.4
Using the same technique, define a script that generates 8 arrows shooting from the robot origin in 8 different directions as shown in the neighboring figure. Hint: Define first a method called arrow: aPoint that draws an arrow pointing in the current direction starting at the given point. Then use a additional variable to remember the origin of the arrow. Once you get done you strongly recommend you to redo the same experiment using the method go: and jump:. This way you will really touch the difference between the two ways of expressing the same problem.



9 Loops and Translations

Before reading the following, try to define the script that draws Figure 1.18.

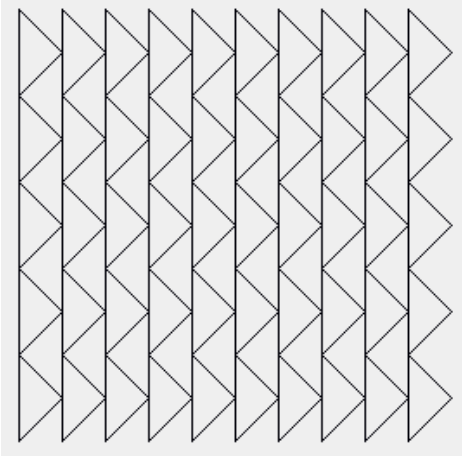
Now we explain how we can produce the drawing shown in Figure 1.18.

Points define a lot of useful methods from which we explain ones we use in the script 1.18: negated and setX:setY:.

- The method negated sent to a point returns a point whose x and y are negated value of the receiving point. Thus, the point (200@400) negated is the point -200@-400. Note that the parentheses are necessary. Indeed, negated is also a method understood by numbers. Thus, the expression 200@400 negated yields the point 200@-400 because the method negated being an unary method is executed before the method @ by the number 400. In the script 1.18 we use this method to produce a translation in the opposite direction.
- The method setX:setY: changes the constituents of a point. Thus, if point is any point, after executing the expression point setX: 200 setY: 400, the point has for x value 200 and for y value 400.

Script 1.18 (*Flying geese cover quilt)*

```
| pica point1 move shift|
point1 := 200@300.
move := 25@0.
shift := -25@50.
pica := Bot new.
5 timesRepeat:
    [ 10 timesRepeat:
        [ pica
            triangleAt: point1
            delta1: 25@-25
            delta2: -25@-25.
         point1 := point1 + move ].
    point1 := point1 + shift.
    move := move negated.
    shift setX: shift x negated setY: shift y ].
```



The script 1.18 uses these two methods within a double loops to generate the flying geese pattern over a large region of the screen. The inside loop is following the spirit of the the script 1.14 except that the orientation of the triangle and that of the translation are rotated so that a line of triangle is now horizontal. The outside loop makes a translation of the last triangle to bring it atop the line of triangle using the point variable shift; then, the translation are reversed so that the next line is drawn in reverse order. The triangles, however, are still drawn with the same orientation. The fact that a second line of triangles appears to point in the opposite direction results from the fact that the two lines of triangles touch themselves. Note that the variable shift must be transformed in a special way: the sign of its x is reversed at the end of each line to compensate for the last translation which is not drawn.

10 Further Experiments

The method translate: aPoint. Defining methods with a precise and simple behavior is a way to simplify your code as explained in chapter **??**. Define the method translate: aPoint . Before looking at the solution we propose in method 1.1, propose an implementation.

Method 1.1

```
translate: aPoint
  "translate the receiver of aPoint x and aPoint y"

  self goTo: (self center + aPoint)
```

Propose a different method translateX: x y: y which takes as argument the value for x and y separately.

Experiment 1.5
Change the definition of the method triangleAt:point2:point3: to use the method translate: aPoint.

Experiment 1.6
Using the method translate: aPoint reimplement some of the methods you created during this chapter and compare their size and complexity.

Summary

- A point is a pair of numbers: it has a x or horizontal coordinates and a y or vertical coordinates.
- 200@400 is a point whose x is 200 and y 400.
- The Smalltalk coordinate system has its origin (0,0) on the top left corner of the screen and y axis goes positive from top to bottom.
- To avoid trouble with points, surround them with parenthesis when they are involved in complex operation.
- goTo: and jumpTo: make the receiver move to the location given as a point.

Message	Description	Example
x @ y	Creates a point of given coordinates	300 @ 600
goTo: aPoint	Ask a robot to move to a given point	pica goTo: 300 @ 600
jumpTo: aPoint	Position a robot to a given point	pica jumpTo: 300 @ 600
point1 + point2	Create a point whose coordinates are the sum of the coordinates of two given points. This is useful to represent translation.	50 @ 200 + 300 @ 600
point1 * number	Create a point whose coordinates are the multiplication of the coordinates of the points and the number	(50 @ 200) *3
point1 negated	Construct a point whose coordinates are the opposite of the original point	(50 @ 200) negated
center	Returns the current position of a robot as a point	barPoint := pica center