Module 4: The System Transcript, Class Point and Inspectors

This module starts by introducing the *System Transcript*, illustrating how it can be used with a number of examples. The Transcript is also used to introduce *Global Variables*, and the TextCollector class is briefly mentioned. The Transcript is also used as a vehicle to introduce *cascaded* expressions, again with many examples.

This module also introduces, as an example, the class Point. This illustrates aspects of the use of *class* and *instance* protocol, the use of instance variables, and the "information hiding" and "encapsulation" aspects of the object–oriented approach. Class Point is used to illustrate the use of *Inspectors*, with many examples.

4.1. Introduction

The System Transcript (or just *Transcript*) is primarily used to display warnings or useful information. For example, when you run an explicit garbage collection operation (by selecting the **Collect Garbage** item from the **File** menu on the Launcher), a message is printed in the Transcript indicating how much memory space was reclaimed, as well as other (possibly) useful information (figure 4.1). Similarly, when you save the VisualWorks image (see module 2), some comments are printed in the Transcript.

Figure 4.1: The System Transcript

The Transcript can also be used as a general–purpose text output area, which is very useful for displaying the results of computations which do not require sophisticated presentation. This module starts by describes how to use the Transcript in this way.

4.2. Using the System Transcript

The System Transcript is like an ordinary Workspace (see module 3), except that it has the additional property of being able to display messages generated from anywhere within VisualWorks. So, the Transcript has the usual Workspace <operate> button menu, and you can type, edit and evaluate expressions just as if it was a Workspace.

You will already have seen a Transcript open on the screen as part of the Launcher when you started VisualWorks. You are advised to have the Transcript open at all times, so that you do not miss any important messages which might be displayed. The inclusion of the Transcript is controlled by the **System Transcript** check–box on the **Tools** menu of the Launcher.

The Transcript is referenced by a global variable Transcript. Global variables can be accessed from any part of the image (see later). The global variable Transcript actually refers to an instance of class TextCollector. The most useful message understood by Transcript (and other instances of TextCollector) is show: ; the argument to this keyword message should be a String. When the object referenced by the global variable Transcript receives the message show: , the argument string is added to the contents of the Transcript.

For example, select and evaluate the following expression in a Workspace, using the **do it** option from the <operate> button menu:

```
Transcript show: 'Hello, world!'.
```

The result is shown in figure 4.2.

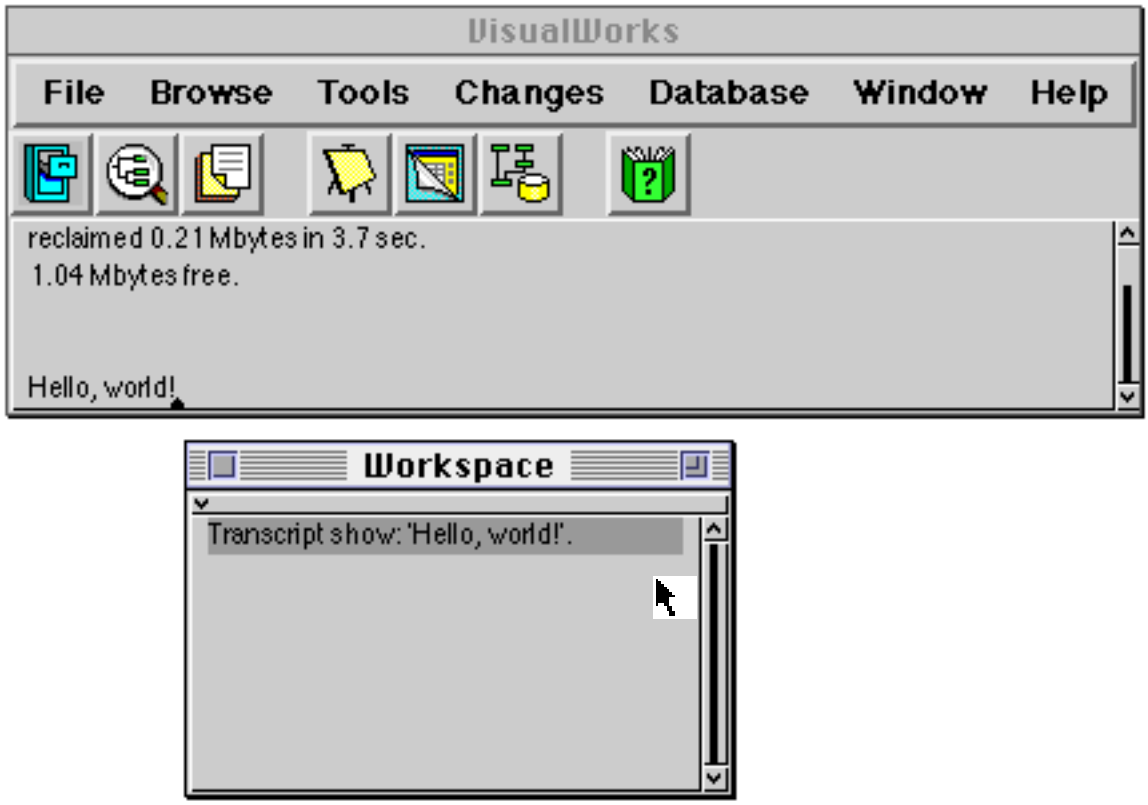

Figure 4.2: Displaying messages using the System Transcript

Another useful message understood by instances of TextCollector is cr, which starts a new line. Other useful messages include space, which inserts a single blank space, and tab, which inserts enough blank space to move to the next tabbing position. The tab message is very useful to allow text output to be lined up neatly — remember that VisualWorks uses proportionally–spaced fonts for

displaying characters, so that it is impossible to line up text output using only spaces.

We've already seen in module 2 that every object in the image understands the message printString. The response to the message is a string containing a suitable printable representation of the receiving object. Since everything in the VisualWorks image is an object, this means that we can print out some representation of anything within the image. The printString method is defined in class Object; the default printing method used to implement this (printOn:) is frequently re–defined in subclasses of Object.

Using Transcript and the printString message together provides a very useful way of printing out the results of computations. For example, if the following expression is selected and evaluated using **do it**, the number 1024 will be printed in the Transcript.

```
Transcript show: (2 raisedTo: 10) printString
```

Omitting the printString message is a very common source of programming errors.

Finally, you should note that the System Transcript only retains the last ten thousand characters inserted into it.

Ex 4.1: Try some further example messages sent to Transcript. You might like to try some of the expressions below:

```
Transcript show: 'Good-bye, World!'
Transcript cr. Transcript tab. Transcript show: 'String on a new line'.
Transcript show: (3+4) printString
Transcript show: (22/7) printString
Transcript cr. Transcript show: (42 raisedTo: 42) printString
```

Ex 4.2 Try getting an instance of SpendingHistory to print itself in the Transcript.

Ex 4.3 Try using the System Browser to browse class TextCollector. Find out what other messages Transcript can respond to. Try out some of these messages. (This class can be found in category Interface-Transcript.)

4.3. Global Variables

The System Transcript illustrates the use of a global variable: a variable name that can be used from anywhere within the image. This is in contrast to the other kinds of variables introduced in module 2: instance variables, which are only accessible from within a particular object, and temporary variables, which are

only accessible within a particular method or block. Another important kind of variable, the class variable, is introduced in module 5.

All global variables start with an initial capital letter, unlike instance and temporary variables. Global variables are usually used to refer to objects which we wish to have a long lifetime. For example, the names of classes (names like Object, Number and so on) are global variables. Several other important objects are referred to by global variables within the image, including the object that controls the Smalltalk processes (Processor — see module 9).

In fact, all global variables within the image are stored in a single table called the *system dictionary*. This is an instance of class SystemDictionary which itself is a subclass of Dictionary (described in module 7). The system dictionary is referenced by the global variable Smalltalk; this means that the global variable Smalltalk appears in the system dictionary referred to by that variable (i.e. is a circularity).

You can look at the contents of the VisualWorks system dictionary by printing its contents in a Workspace (remember that every object in the image can be printed out). Selecting and evaluating (using **print it**) the following expression is one convenient way of doing this.

```
Smalltalk keys asSortedCollection
```

This will display the names of every global variable in the image, including the names of all the classes. You can also more conveniently inspect the contents of the VisualWorks system dictionary using an Inspector (see later).

Global variables are usually declared simply by typing and “accepting” the name of the new variable (with an initial capital letter, of course). If the variable does not already exist, a Confirmer will appear, asking what kind of variable is required. You should select the **global** option (see figure 4.3). You could also select the **Correct It** option, if you had mis–typed the variable name.

Alternatively, the new global variable name, together with the object to which it refers, can be inserted directly into the system dictionary, using an expression like:

```
Smalltalk at: #NewGlobal put: (55/7).
```

Global variables can be removed using the following expression:

```
Smalltalk removeKey: #NewGlobal.
```

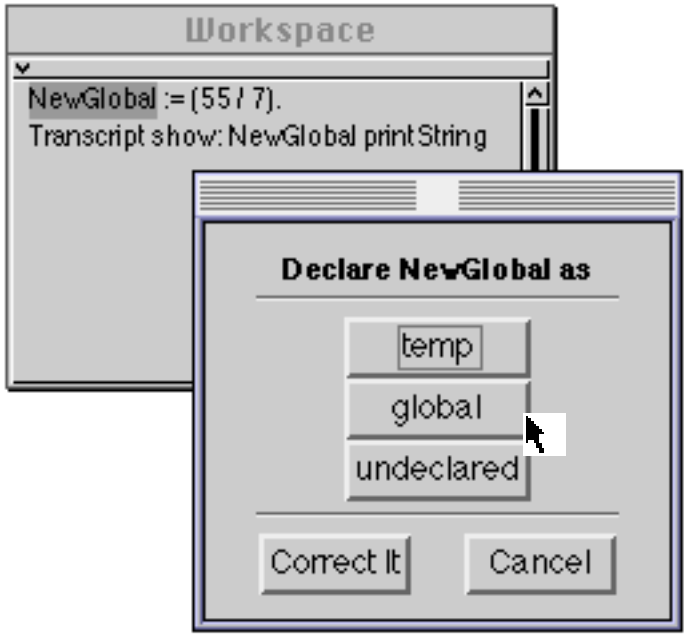

Figure 4.3: Declaring a Global variable

It should be pointed out that new global variables are relatively rare in VisualWorks applications, and extensive use of globals suggests that the structure of the application has not been well thought out. A global variable should only be used if you are quite sure that there should never be more than one object with these particular properties.

Ex 4.4: Look at the contents of the system dictionary Smalltalk, by printing it out in a Workspace (or in the Transcript).

Ex 4.5: Try creating new global variables, in both of the ways described in this section. Also, try removing the global variables you have created.

Ex 4.6: What happens if you try and declare a global variable which does not start with an upper–case letter? Try it and find out.

4.4. Cascading Expressions

You will have seen in the examples and exercises earlier in this module how it is frequently necessary to send several messages in sequence to the System Transcript. In general, repeated message sends to the same object occur quite frequently. To aid this, a syntactic form called “cascading” is provided. This uses a different separator character ‘;’ (semi–colon).

For example, the following sequence of expressions might be used:

```
Transcript cr.
Transcript show: 'The result of 6 times 7 is'.
Transcript tab.
```

```
Transcript show: (6*7) printString.
Transcript cr.
```

A series of messages (cr, show:, tab and so on) is sent to the same object (Transcript). Clearly, the repeated use of the global variable name Transcript involves tedious repeated typing.

These expressions could be re–written using a cascade, to avoid using the Transcript identifier quite so often:

```
Transcript cr ;
    show: 'The result of 6 times 7 is' ;
    tab ;
    show: (6*7) printString ;
    cr.
```

You should be able to see that exactly the same sequence of messages has been sent. In both cases, five messages are sent to the Transcript. (Here the message expressions have been spread over several lines for clarity — it is not necessary to do this in practice.)

The use of cascade expressions frequently results in fewer, shorter expressions, with fewer temporary variables being used. However, any cascaded expression can be re–written as a sequence of message–sends without cascades, possibly with the addition of temporary variables. In some cases, the expressions may be much easier to understand in a form without cascades.

It is important to stress the difference between *cascaded* and ordinary *concatenated* message sends. Consider the following two expressions:

```
receiverObject message1 message2.
receiverObject message1 ; message2. "Note the cascade."
```

In the first expression, receiverObject receives message1, and evaluates the appropriate method. This method answers with another object; it is this *new* object that receives message2. In the second case, receiverObject receives message1 as before and evaluates the corresponding method. However, this new object is *discarded*, and it is receiverObject again which then receives message2. The two expressions are equivalent only if message1 answers with self; i.e. receiverObject itself.

Ex 4.7: Try rewriting some of the Transcript examples from Ex 4.1 using cascades.

Ex 4.8: The following expressions create an Array containing three strings using a sequence of at:put: messages. Re–write these as three expressions using cascades, keeping the temporary variable array.

```
| array |
array := Array new: 3. "Create a new Array, length 3."
array at: 1 put: 'first'. "Put a string in the first location."
array at: 2 put: 'second'. "Put a string in the second location."
array at: 3 put: 'third'. "Put a string in the third location."
Transcript cr. "New line in the Transcript."
Transcript show: array printString. "Print the array in the Transcript."
```

Ex 4.9 (Much harder.) Try rewriting the above as a single expression, but with all the same message sends, removing the temporary variable array. **Hint**: you may need to use the message yourself, implemented in class Object.

4.5. Class Point

Class Point represents the abstract notion of locations in a *two–dimensional plane*. This is a particularly useful idea, especially when we are interested in manipulating objects on a (two–dimensional) display screen. Points are very widely used within VisualWorks, particularly in conjunction with rectangles (see module 6).

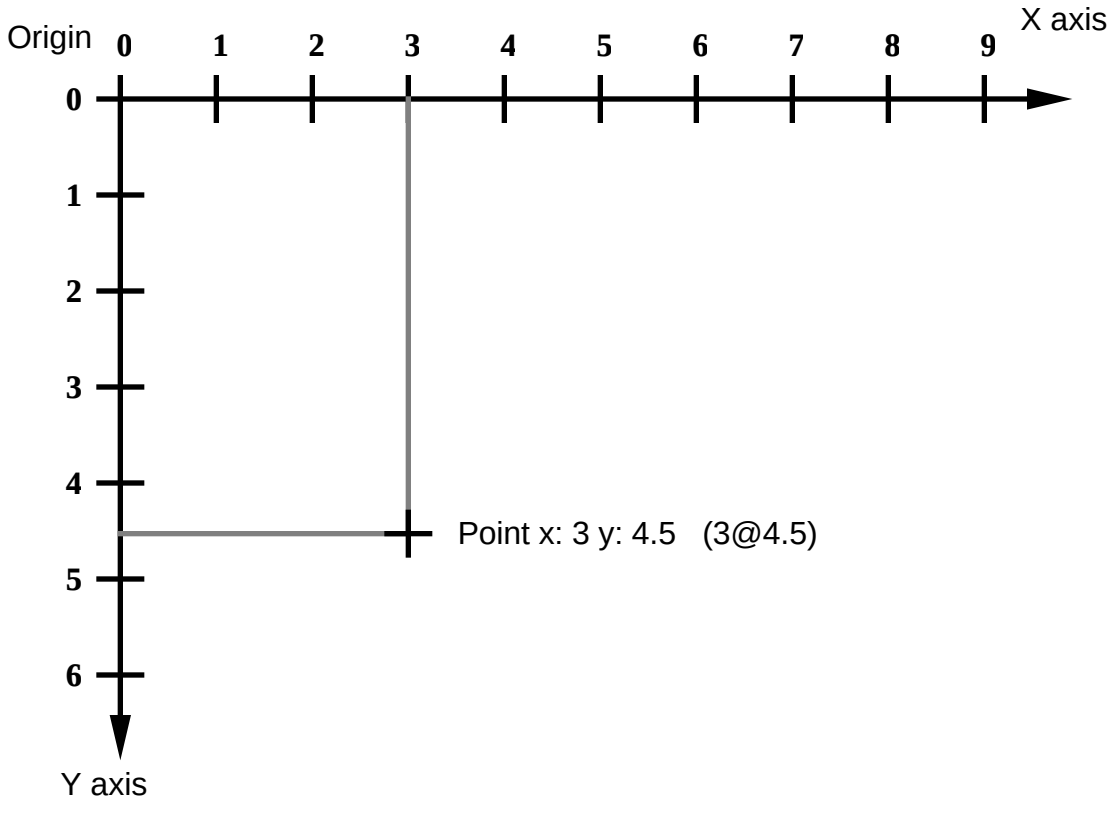

Figure 4.4: The VisualWorks co-ordinate scheme

Internally, a Point is represented in cartesian (rectangular) co-ordinates, although other representations are possible (see later). Two instance variables are defined, x and y, giving the displacement from the origin in the horizontal and vertical directions respectively. Unusually, the co-ordinate scheme is left–handed (see

figure 4.4), so that, while the x–axis runs left–to–right, the y–axis runs top–to–bottom. (Conventionally, the y–axis runs bottom–to–top.) This is because VisualWorks is frequently concerned with the display of text — usually displayed left–to–right and top–to–bottom.

4.5.1. Creating a new Point

Sending the message new to a class will usually result in a new instance of that class. When the class Point receives the message new, the corresponding method returns a new instance of Point with both x and y instance variables uninitialized — i.e. each with a reference to the undefined object nil (see module 2). In other words, a Point is created which represents nowhere in particular. We will then have to define the instance variables by sending further messages to the newly–created instance. What we really want is a way of creating initialised instances of class Point — i.e. with their instance variables suitably defined.

An instance creation class method is already available by which initialised instances of Point may be created. This method is called x: y:, and allows new instances to be created with the x and y instance variables initialised by the argument numbers, for example:

```
Point x: 3 y: 4.5.
```

or,

```
Point x: 2 y: 3.
```

This is a slight improvement; it is now easier to create initialised instances. However, since points are so widely used, a shorthand way of creating points is provided. The message @ (a binary message selector) is understood by instances of subclasses of Number (see module 6). This answers with a new instance of Point, created from the receiver (for the x–co-ordinate) and the argument (for the y–co-ordinate). This means that points can be expressed simply as 2@3. This mechanism for creating new points using @ is so widely used that the same format is adopted when points are printed.

Class Point has a large number of methods available. You are advised to spend some time browsing this class. The instance protocols provided include:

4.5.2. Accessing

The current values of the x and y instance variables can be accessed, using the messages x and y respectively. The corresponding methods (also called x and y) simply return the current value of the appropriate instance variable. Similarly,

the instance variables can be set using the x: and y: messages. You should note that the relationship between the instance variable names x and y, and the method names x and y is simply one of convenience; there is no *a priori* reason why they should have the same names. However, giving the same names to instance variables, and to methods that access those instance variables (often called simply *access methods*) is conventional, and frequently used.

4.5.3. Comparing

Methods are provided to compare Points for equality (=), and various kinds of inequality (<, >, <=, >=, ~=, and so on). For example, a Point is "less than" another Point if it is both above, and to the left of the first Point; i.e. closer to the origin in both co-ordinates.

4.5.4. Arithmetic

All the usual arithmetic operations (+ and so on) are defined on Points. For example:

```
(3 @ 4.5) + (12.7 @ -3)
((22 / 5) @ 14) - (2 @ 13)
(3 @ 4) * (2 @ 2)
(99 @ 100) / (4 @ 4)
(-14 @ 13.95) abs
```

These methods also work if the argument is a scalar (any kind of number), rather than a Point. Examples:

```
(3 @ 4.5) + 12
(3 @ 4) / (22 / 7)
```

4.5.5. Truncation and Rounding

The rounded method answers with a new Point with x and y values converted to the nearest integer values. The truncate: method answers with a new point with x and y values truncated so that they lie on a grid specified by the argument (another Point).

4.5.6. Polar Co-ordinates

The r method answers with the distance of the Point from the origin. The theta method answers with the angle (in radians) from the x–axis. These methods allow locations to be converted to polar co-ordinate form.

4.5.7. Point Functions

Several useful methods are provided in this protocol. These include: dist:, giving the absolute distance between the receiver and argument Points, and transpose, which answers with a new Point with x and y swapped over.

There are several other instance protocols provided, which are not considered here.

Ex 4.10: Try creating various instances of class Point, using each of the instance creation methods mentioned above. Use the show: and printString messages to display the values in the Transcript.

Ex 4.11: Browse class Point; this class can be found in category Graphics-Geometry. Using the System Browser, find out the effect of dividing a Point by an integer. Type in and evaluate (using **print it**) an expression to find out if your answer is correct.

Ex 4.12: Try out some of the methods defined in the point functions and polar co-ordinates protocols. For example, find out the result of each of the following expressions:

```
101.7@77.1 grid: 4@4.
43@17 dist: 45@103.
(4@3) r.
(4@3) theta.
```

Ex 4.13: (Harder.) The dist: method in the point functions instance protocol answers with the absolute (positive) distance between the argument and receiver Points. This is the length of a straight line joining these two Points. In manhattan geometry, motion is not permitted in arbitrary directions, but must follow horizontal and vertical lines only. (This is just like travelling in a modern city, laid out as a series of "blocks" — hence the name!). Write a new method called manhattan:, in the point functions instance protocol of class Point, which answers with the absolute distance between the receiver and argument Points, when travelling only in horizontal and vertical directions.

4.6. Alternative Representations for Class Point

It is worth observing at this point that the internal representation of class Point (in cartesian co-ordinates) is not the only possible way in which locations in two–dimensional space can be specified. For example, in a polar co-ordinate representation, a location is specified as a distance ('r') from a defined origin, together with an angle ('theta') from a defined axis through that origin (see figure 4.5).

It would be perfectly feasible to implement class Point so that each instance had instance variables r and theta. When the message x was sent, for example, the corresponding method would have to compute the appropriate value from r and theta. However, methods such as r could be implemented simply to answer with the value of the corresponding instance variable. All the other methods

currently implemented by class Point could be re–implemented using the new instance variables.

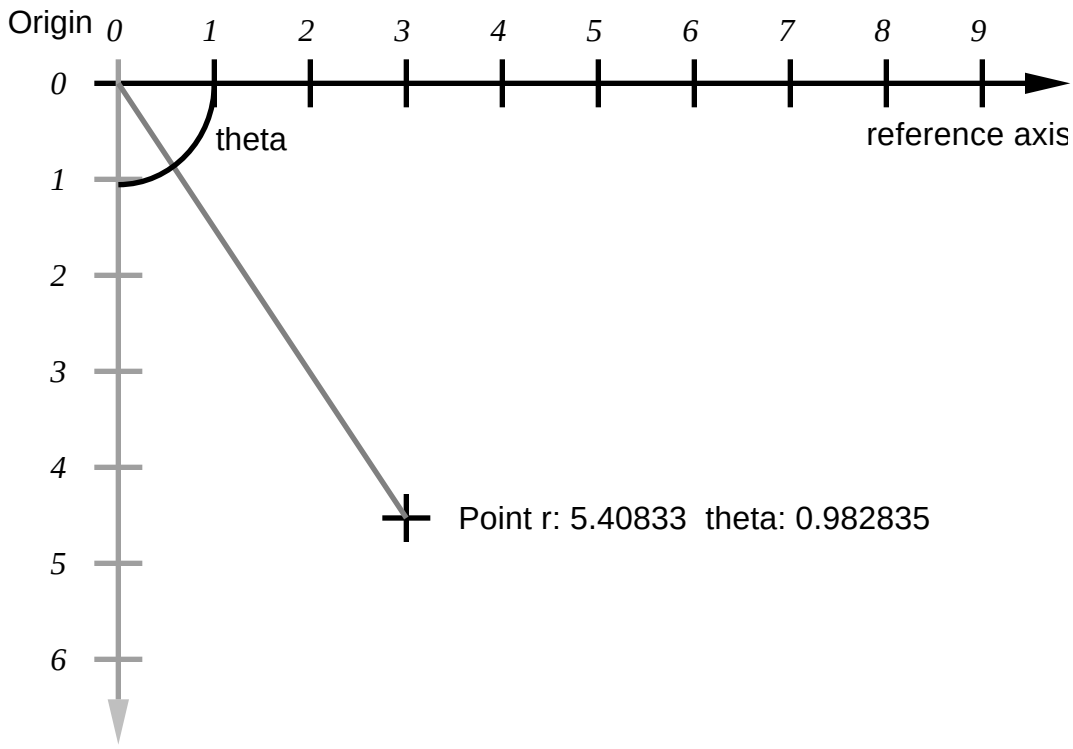

Figure 4.5: Alternative representation for class Point

Thus, it is possible to implement class Point in a completely different way but, provided that the same methods were implemented to give the same result, there would be *no change* as far as any other object in the image is concerned. This illustrates the information–hiding features provided by an object–oriented system.

Class Point is actually implemented using x and y instance variables for performance reasons. As points are most frequently used to describe rectangular areas (such as panes on the screen), the cartesian operations are the ones most frequently used.

Ex 4.14: Implement a new class NewPoint that behaves just like Point, but using a different internal representation as suggested above.

4.7. Inspecting Instances of Classes

We have already investigated the use of Browsers to view the source code of the methods associated with various classes. The effect that a message sent to an instance of some class can be determined by examining the appropriate method. However, the only way we have so far discovered to find out the state of a particular instance is to print it out (in the Transcript, or using **print it** from the

<operate> menu). This is clearly less than satisfactory, and we need a better way of viewing the internal state of an object.

All objects understand the message inspect. This is implemented in class Object to open a new kind of window called an *Inspector* on that object. For example, to inspect an instance of class Point, the following expression can be used:

```
(Point x: 3 y: 4.5) inspect
```

Since objects are inspected so frequently, an **inspect** option is provided on the <operate> menu associated with Workspaces and other text editing windows. In either case, an Inspector is opened (figure 4.6).

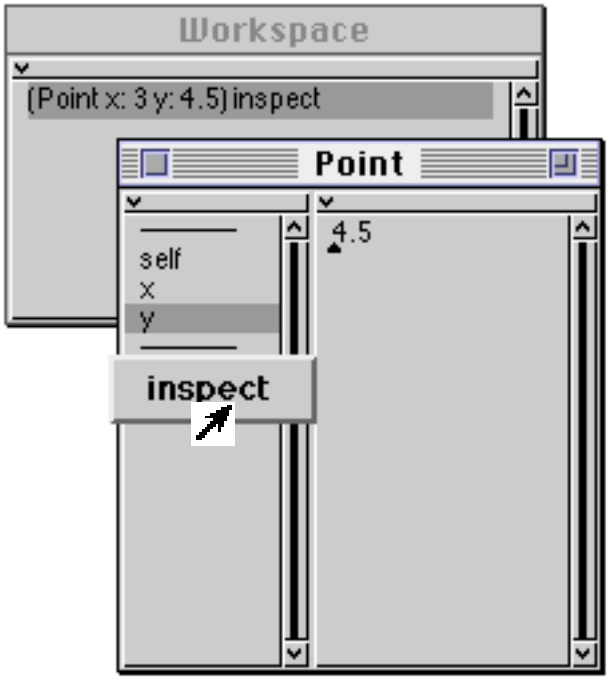

Figure 4.6: Inspecting an instance of class Point

An Inspector is labelled with the class of the object being inspected, and consists of two panes. The left–hand pane is a list of the instance variables of the object (like the lists in the top part of a System Browser) plus the pseudo–variable self, representing the actual object being inspected. One of the items can be selected from this list; the right–hand pane, which is an ordinary text pane (like a Workspace), displays the current value of that instance variable. In this way, we can inspect any object in the image in some detail.

The left–hand pane has an <operate> menu; this has one item (**inspect**), which allows the selected instance variable to be inspected; another Inspector is

spawned on that object. This allows complex structures of interrelated objects to be explored.

Since the right–hand pane of the Inspector is a Workspace, we can select and evaluate expressions in the usual way. However, we can also write expressions which use the named instance variables, and the pseudo–variable self (see figure 4.7). We say that the expressions we select and evaluate are evaluated “in the context of the object being inspected”.

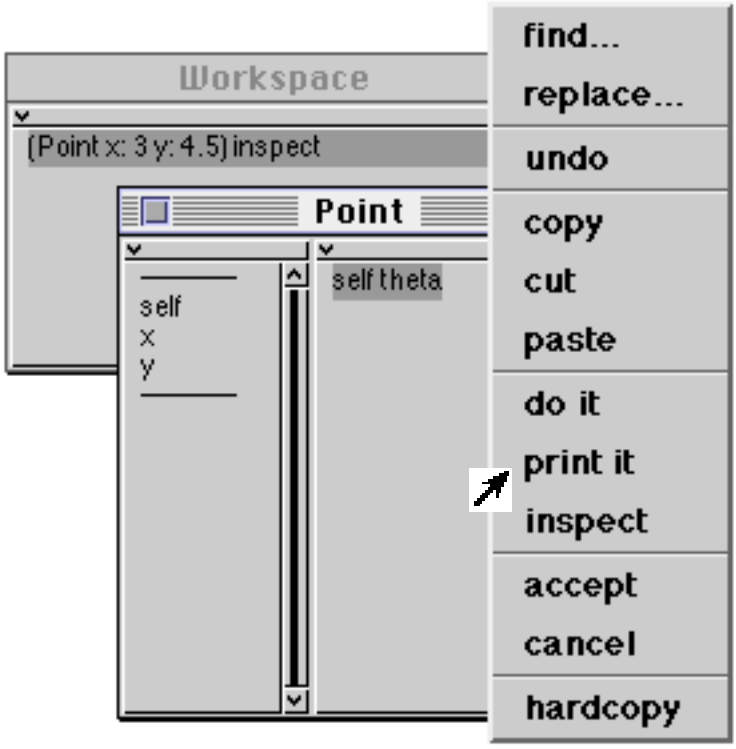

Figure 4.7: Evaluating an expression using self, in the context of the inspected object

As well as being able to view the values of instance variables of any object, Inspectors also allow us to modify these values. Any expression can be typed into the right–hand pane of an Inspector; if the **accept** option is selected from the <operate> menu (figure 4.8), then the resulting object is used as the new value of the instance variable (figure 4.9). You should note that you cannot change self in this way.

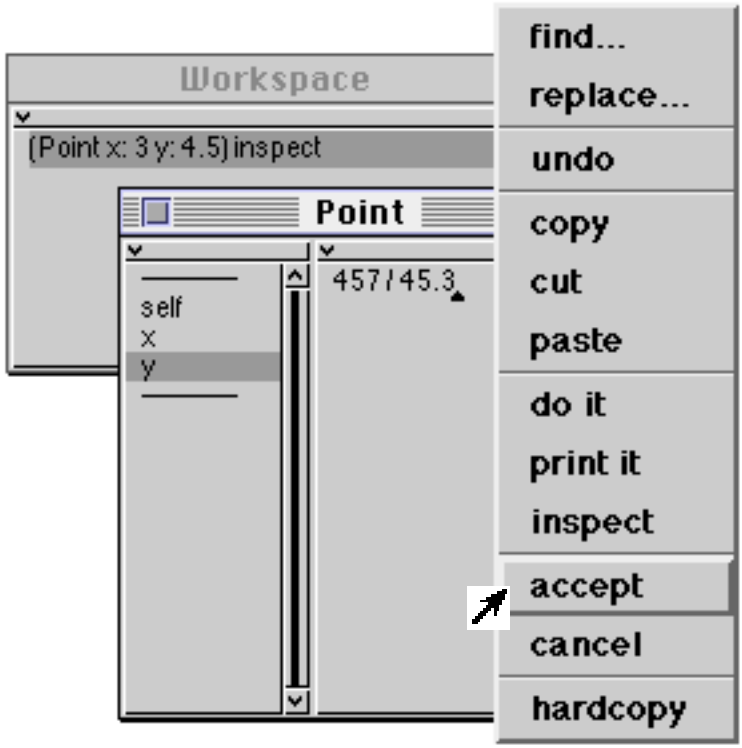

Figure 4.8: Changing the value of a Point's instance variable, using an Inspector

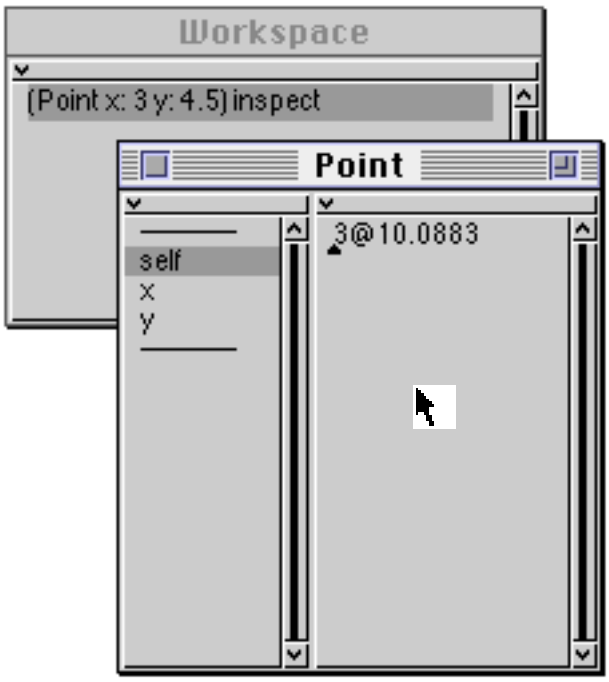

Figure 4.9: The result of changing an instance variable using an Inspector

It should be pointed out that Inspectors allow direct access to the instance variables of the object being inspected, and therefore deliberately break the

information–hiding notion which is central to object–oriented programming. Thus, they should be used for testing and debugging purposes only.

Ex 4.15: Create and inspect various instances of class Point, in the ways suggested above. Look at the values of the instance variables. You might also like to try inspecting other objects you already know about. Experiment with the **inspect** option from the <operate> menu in both the left–hand and the right–hand panes of the Inspector.

Ex 4.16: Try evaluating some expressions using the values of instance variables, or self (as in figure 4.8, for example). Also, try modifying the value of an instance variable, by typing an expression and using the **accept** menu item.

4.8. Other Inspectors

A small number of special Inspectors are provided for instances of certain classes. In particular, Inspectors are implemented for instances of class OrderedCollection (see module 8) and its subclasses (figure 4.10), as well as instances of class Dictionary (module 7) and its subclasses (figure 4.11). Both these Inspectors have extra items on their <operate> menus.

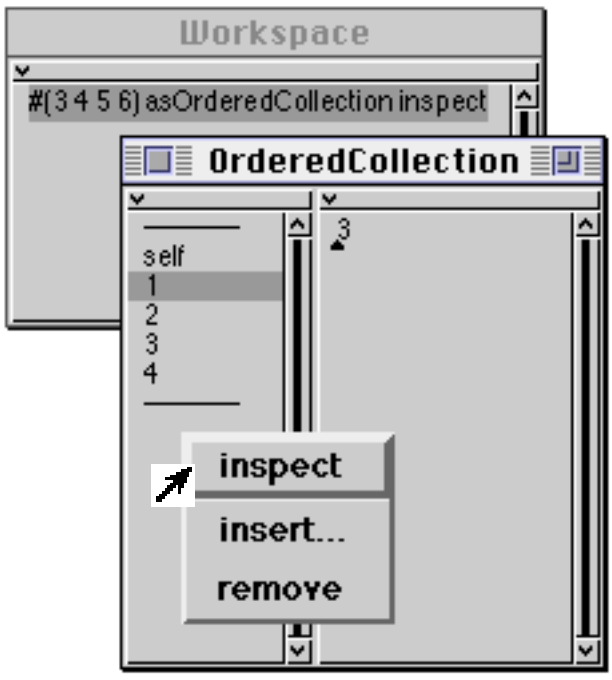

Figure 4.10: Inspecting an OrderedCollection

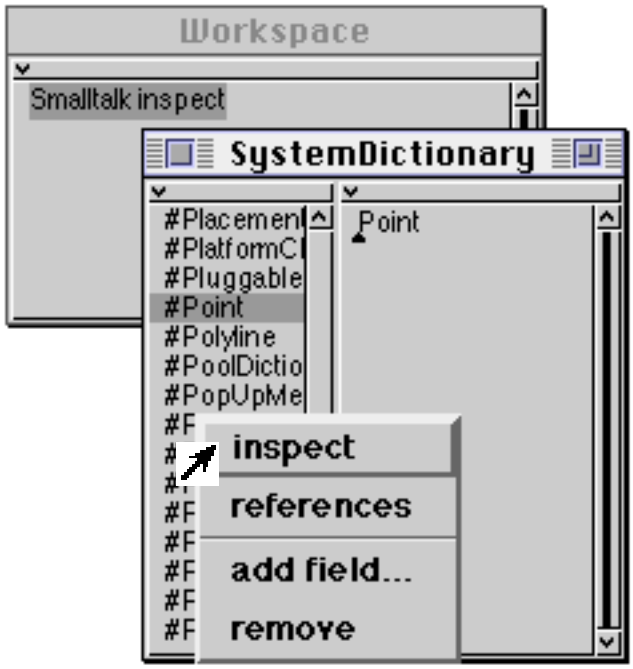

Figure 4.11: Inspecting the System Dictionary

Ex 4.17: Try inspecting the system dictionary Smalltalk, which contains all the global variables in the image. **Warning**: be very careful not to remove anything from this dictionary!

Ex 4.18: You might like to try inspecting a class, to find out its internal structure.