Chapter 3 - Principles of Smalltalk

Overview

After dedicating the first two chapters to the general ideas of object-oriented programming and program development, we will now shift our attention to Smalltalk and explain its main rules and concepts including the rules of its syntax and code evaluation.

We will also introduce the use of the Inspector, the Debugger, the Workspace, and the Transcript. The purpose of the Inspector is to view the components of an object. The Debugger is used mainly to catch and trace errors in execution. It can execute Smalltalk statements message by message and provides an insight into message evaluation. The Workspace is used to test pieces of Smalltalk code before they are incorporated in methods. The Transcript is used by the environment itself to display system messages but Smalltalk programmers often use it for intermediate output during testing.

3.1. Basic rules

In this section, we will use *Farm* and *Pen* worlds to show what Smalltalk programs look like, and introduce several basic rules of Smalltalk.

Example 1. Smalltalk version of *Farm* commands

As you already know, executing an operation requires selecting a receiver and sending it a message. You have already practiced this art in *Farm 1*, *Farm 2*, and *Farm 3*. *Farm 4* shows you how you could achieve the same effect using Smalltalk – although this version of the code is still somewhat preliminary. The generated code and the use of the program s self-explanatory: Can you say which sequence of actions produced the code shown on the right hand side of Figure 3.1?

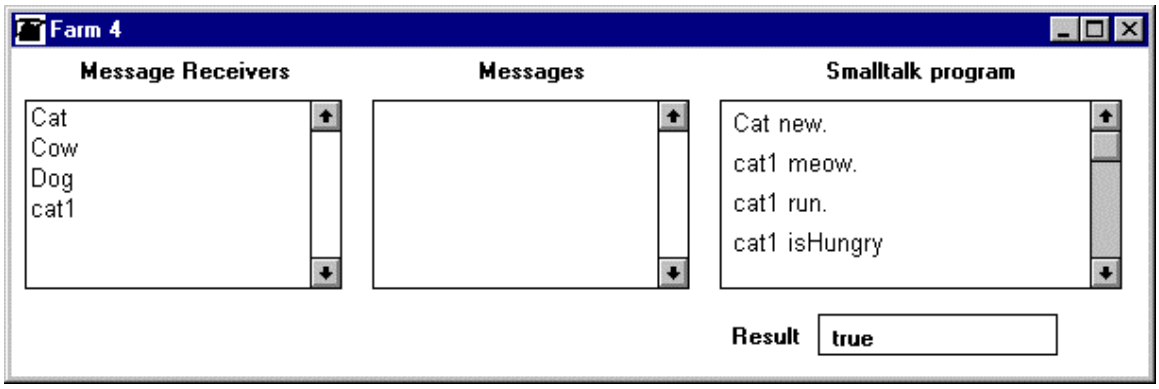

Figure 3.1. *Farm 4* displays Smalltalk equivalents of commands used in the lower level versions of *Farm*.

Pen world is another testing ground for Smalltalk code but unlike *Farm* which is too trivial, *Pen world* also allows you to solve simple problems based on drawing lines and erasing them. Figure 3.2 is an example of the *Pen world* user interface and we will immediately proceed to use it to solve a few simple tasks and examine the corresponding code. We will remind you that you should execute

```
PenWorld open
```

to run *Pen world*. In this example, we will use *Pen World 3*.

Example 2. Draw a square

Problem: Draw a 50-pixel side square whose lower right corner is the home position of the pen (Figure 3.2).

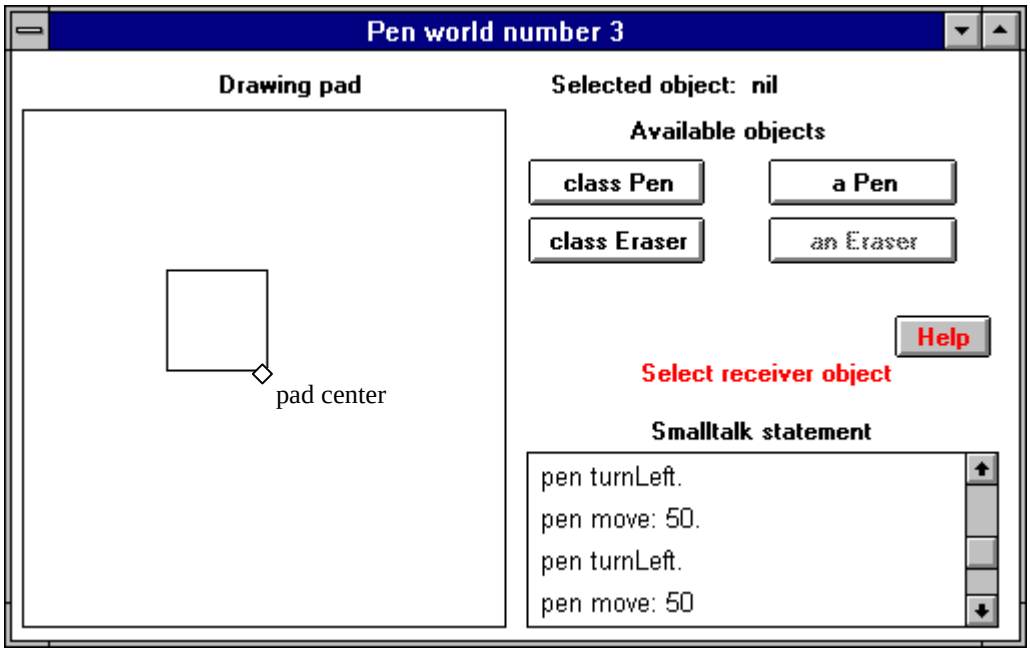

Figure 3.2. *Pen world 3* after executing Task 1. Only the last few statements are shown but the text can be scrolled to show all statements executed so far.

Solution: To be able to solve the problem, we must understand the mechanics of the *Pen World*. First, before we can use a pen or an eraser, we must create one using the Pen or Eraser class. When Pen creates a pen, it leaves it in the 'down' position in the center of the drawing pad and 'touching the drawing surface', ready to draw. It is oriented towards the top of the drawing pad so that when you move it, it will draw a straight line going up. The functions available through the button interface include lifting and lowering the pen, turning it left or right by 90 degrees, and moving it by a specified number of pixels (screen positions) in the current direction. Additional messages are available through the programming interface in *Pen World 5*.

As you execute the task, the text view displays a Smalltalk *statement* each time you click an *object selection - message selection* pair, and when you are finished, the text view contains the following self-explanatory code fragment:

```
Pen newBlackPen.
pen move: 50.
pen turnLeft.
pen move: 50.
pen turnLeft.
pen move: 50.
pen turnLeft.
pen move: 50
```

Several notes are in order:

1. A *statement* consists of a receiver followed by a message. (This is a preliminary definition.)
2. Smalltalk programmers informally use the term *code fragment* for a sequence of Smalltalk statements written to test a programming idea. We also use code fragments to illustrate concepts and the use of existing objects, when creating new classes or methods would require too much space.

3. The term *program* is sometimes used with the same meaning as *code fragment* even though it may create the wrong impression that Smalltalk applications are big code fragments. As you know, Smalltalk applications consist of classes, with code organized into methods rather than code fragments so a code fragment is only a preliminary stage of Smalltalk programming before code crystallizes into methods. In fact, the idea of a code fragment is so foreign to Smalltalk, that before you execute one, Smalltalk converts it temporarily into a method and executes it as a message. The only real difference between a code fragment and a method therefore is that code fragments Smalltalk does not store code fragments as a part of a class. Also, whereas code fragments are usually created in a Workspace, classes are created with the browser.
4. In a sequence of statements, individual statements are separated by periods. A period is not required behind the last statement.
5. Names (*identifiers*) of Smalltalk messages and objects consist of a sequence of letters or digits and start with a letter. There is no limit on the length of an identifier and Smalltalk treats all characters in an identifier as meaningful. Examples of *legal* identifiers are aPen, PenWorld1, RedPen, aRedPenWithAThickTip, pen1, drawCircle, draw3Circles. Examples of *illegal* identifiers are three%, 3pens, this&that, and pen 1. Deviations from the naming rule may produce unexpected results. As an example, if you try to call an object red pen, Smalltalk will think that red is the receiver and pen a message.
6. Names of *classes* start with a capital letter as in Pen or Eraser; names of *instances* start with a lower case letter as in pen or eraser. Names of *all messages* should start with a lowercase letter.
   Smalltalk identifiers are *case sensitive* which means that two identifiers are different even if they differ only in the case of some characters. For example, redPen and redpen are different identifiers, and turnLeft and turnleft are also different. Using the wrong case happens so often that Smalltalk has a simple built-in spelling corrector: If you misspell an identifier in a program and try to execute it, Smalltalk will display the word and ask you if you want to attempt automatic correction. If you do, it will find several similar identifiers that *are* defined in the library or in your program, let you select one, and make the replacement automatically. If you misspell the case of an identifier or make a similar minor mistake, the corrector will be able to offer a solution but for more serious spelling mistakes the corrector is insufficient and you must make the correction manually.
7. To make programs easier to understand, use descriptive identifiers. As an extreme example showing the result of a very poor choice of identifiers, consider the following version of the program above in which we chose legal but meaningless identifiers:

   ```
   XY z.
   a f: 50.
   a t.
   a f: 50.
   a t.
   a f: 50.
   a t.
   a f: 50
   ```

   While this program would work if all the identifiers were defined, the original version gives a good idea of what is going on but this version does not give any indication of what the program does. This is an extremely important consideration because readability greatly influences code maintainability, and maintenance is the most expensive part of software development. Readability is improved by following conventions that other Smalltalk programmers use and we listed the most important ones in Appendix 5.
8. Smalltalk programmers often combine several words into a single identifier and capitalize the start of each word inside the name as in newBlackPen or turnLeft.
9. The rules of Smalltalk make it easy to understand the structure of messages. As an example, consider the statement

   ```
   Pen newBlackPen
   ```

Pen is the receiver because it comes first, newBlackPen is a message. The name Pen begins with a capital letter which suggests that it refers to a class[1]. This being the case, newBlackPen is a *class message*. As another example, pen in

```
pen move: 50
```

is the receiver and move: 50 is a message whose argument is 50. The name pen starts with a lowercase letter so it refers an instance of a class, and this means that move: is an instance message. The argument specifies that this move should displace the pen by 50 steps.

Main lessons learned:

- A Smalltalk statement is a receiver-message pair (preliminary definition).
- Names of objects and messages are called identifiers.
- Smalltalk rules for identifiers are as follows:
    - All identifiers must begin with a letter and the rest must consist of letters or digits.
    - Identifiers are case sensitive and have unlimited length.
    - Names of classes and class variables start with a capital letter, names of instances, instance variables, and all messages start with lowercase letters.
- A code fragment is a sequence of statements written to test a programming idea. The term program is sometimes loosely used for the same purpose although a Smalltalk program-application is a collection of classes.

Exercises

1. Write a code fragment to draw the rectangle from Example 2 in counter clockwise direction. Then execute the task in *Pen world 3* and check your answers against the automatically produced code.
2. Which of the following are not legal identifiers and why? newbook, squared, Book, aBook, new-book, newbook, new_book, negated, correct%, book@shelf, squareRoot, account, 3times, Account, factorial, display, display area.
3. Which of the legal names in the previous exercise are probably names of classes, and which ones are names of instances? Which ones are probably names of messages? Note that the only safe way to decide whether an identifier denotes a message or an object is to see it in the context of Smalltalk code.
4. Suggest descriptive Smalltalk identifiers for the following objects: a cash register class, a cash register instance, a book object in a library catalog, a green car, a passenger car class.
5. Suggest descriptive Smalltalk identifiers for the following messages: Dispense cash, check out a book, create a new object, turn left twice, close a window on the screen. See Appendix 5 for guidelines.
6. Use the System Browser to find five well selected names in the instance protocol of class Character (category Magnitude-General).
7. Is there a difference between names of class messages and names of instance messages? (Hint: Use the class setting in the Browser and check out a few messages.) If not, how can you tell whether a message is an instance message or a class message?

3.2 Maintaining access to objects - variables

If your drawing pad contains one pen and you want to send it a message, there is no ambiguity as to which pen you mean. But what if you have more pens? In a *Pen world* controlled by buttons, the solution is easy - choose the pen by clicking its image. However, if you want to control the pen by Smalltalk code, you need a different name for each pen. As we already mentioned, identifiers that denote objects are called

[1] Use this reasoning with caution since names of some other kinds of objects such as class and global variables also begin with a capital. However, in most cases a capitalized name is a class name.

variables because the state of the object attached to a variable may change during execution. In *Pen world 4*, we added variables to the displayed code and the code that *Pen world 4* produces is complete and correct Smalltalk code.

Task 2. Draw two squares

Use *Pen world 4* to draw a red square with a 70-pixel side, and a blue square with a 90-pixel side (Figure 3.3). The origin of the red square (its upper left corner) is at offset 30@50 (30 right and 50 down) from the home position at the center of the pad, the blue square's origin is the home position. Draw the squares using the pens alternately.

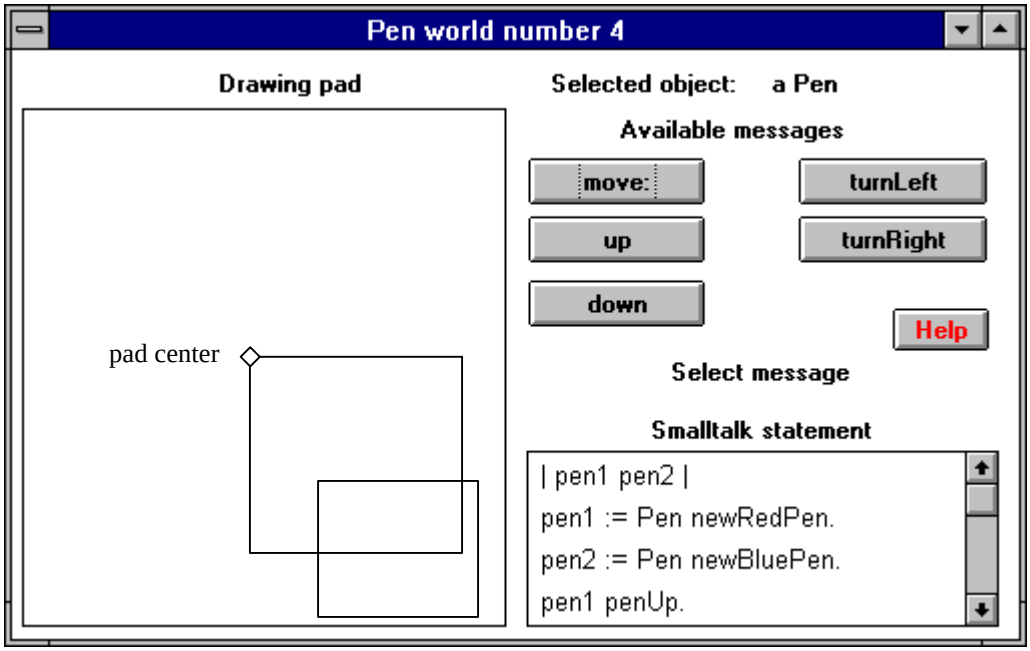

Figure 3.3. Example 2 executed in *Pen world 4*.

The algorithm to solve the problem is as follows:

1. Create two pens, one red and one blue.
2. Lift the red pen and move it to the desired origin of the red rectangle.
3. Lower the red pen.
4. Draw the rectangles using the pens alternatively.

When you execute this algorithm using the pen buttons in *Pen world 4*, the text part of the window (the text subview) will display the code listed below except for the *comments* (the text surround by double quotes) that we added to relate the code to our algorithm.

```
| pen1 pen2 |
"Create two pens"
pen1 := Pen newRedPen.
pen2 := Pen newBluePen.
"Position the red pen without drawing, get it ready to draw"
pen1 up.
pen1 turnRight.
pen1 move: 30.
pen1 turnRight.
pen1 move: 50.
pen1 down.
```

```
"Draw the squares, alternating the pens"
pen1 move: 70.
pen1 turnLeft.
pen2 turnRight.
pen2 move: 90.
etc.
```

The first line of the code fragment is *variable declaration*. The names of all Smalltalk variables used in a method or code fragment must be declared before the first statement. As you can see, a variable declaration consists of a list of identifiers separated by blanks and surrounded by vertical bars.

If you were *writing* the program (rather than executing it by *clicking* buttons), the choice of the name would be completely up to you and you might prefer to call the pens redPen and bluePen; *Pen world 4* creates variable names automatically - and without much intelligence.
The line

```
"Create two pens"
```

is a comment. Programmers use comments for documentation and Smalltalk ignores them. A comment is any text surrounded by double quotes and you can put it anywhere you want except inside identifiers and other elementary parts of the language such as numbers. Use comments to explain the purpose of a method and to clarify less obvious code.

Immediately after variable declaration and just before the first statement, the code has not yet attached variables to any objects and its value is thus undefined. Since a Smalltalk variable must always reference an object, each variable is initially bound to a special object called nil. Object nil is the sole instance of class UndefinedObject that exists mainly for this reason.
The line

```
pen1 := Pen newRedPen
```

is called an *assignment* because it assigns an object to a variable (Figure 3.4). In this case, it binds a new blue pen to variable pen1. In the rest of the code fragment, every reference to pen1 refers to the instance of Pen created on this line and this binding persists even when the state of the pen changes.

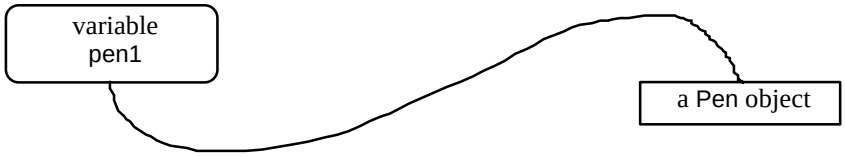

Figure 3.4 Assignment pen1 := Pen newRedPen binds variable pen1 to a Pen object

The only way that a variable's binding can change is that another assignment binds it to a different object which is allowed but not recommended. Objects that are bound to a variable declared in a method or a code fragment remain active while the code is active whereas object that are not bound to variables or other objects cease to be valid after they cease to be referenced. The *automatic garbage collector* of Smalltalk then automatically removes such unreachable objects from memory[2].

As you can see, an assignment consists of the name of a variable followed by the *assignment symbol* := (an approximation of a left pointing arrow ← to indicate the direction of the assignment), and an expression that calculates the object being assigned. Note that there is no space between : and =. The usual ways of reading an assignment such as

```
pen := Pen newBluePen
```

[2] Appendix 8 contains more information about garbage collection.

is ‘pen *gets* (or *refers to*, *is bound to*, *gets the value of*) Pen newBluePen‘. Note that there is no space between : and =. An assignment is one of the forms of an expression and as such it returns a value – its right hand side.

After these language-related issues, note one other thing about the code: If you did not click the buttons in the same order as we did, you ended up with a program in which the same statements are executed in a different order, or maybe even a program using different statements if you used a completely different strategy. Yet, the resulting drawing could be the same. This shows that the same result can often be obtained in many different ways. Sometimes all solutions are equally satisfactory, at other times one solution may be better than another.

One should, of course, strive for the best solution but what is the best solution may depend on the context. You might expect that the best solution is the one that executes fastest or the one that requires the least computer memory. This is true in some cases but in most situations the best solution is the one that makes most sense and is reasonably efficient in terms of execution speed and use of memory. This is because the most important considerations in program development usually are how fast the application can be developed, how much it costs to develop, and how difficult it is to maintain. The cost of maintenance of a commercial product is normally higher than the cost of development and because maintainability is closely related to understandability, clarity is a paramount consideration. Even when code must run as fast as possible or use as little memory as possible, expert programmers write a readable solution first, and optimize its performance after getting the program to work, when they are certain that the overall approach is correct.

Main lessons learned:

- A variable is an identifier bound to an object. We use it to access the object in a program.
- Smalltalk recognizes temporary variables used in code fragments and methods, and instance and class variables holding an object’s internal state. Temporary variables maintain access to an object within a code fragment or a method whereas instance variables are attached to an object.
- Declare a variable when you need to refer to an object several times in a code fragment or method. Objects that are not referenced several times do not need to be assigned to a variable.
- Temporary variables may also be useful to explain the logic of complex expressions.
- All temporary variables must be declared at the beginning of the code fragment or method.
- All variables are initially bound to the nil object, the single instance of class UndefinedObject.
- An assignment assigns an object to a variable. It creates a binding between the identifier and the object.
- An object remains bound to a variable until a new assignment binds the variable to another object.
- An object bound to a variable persists as long as the variable is valid: A temporary variable is valid within the code fragment or method in which it is declared, objects bound to instance variables persist as long as the object to which the variables belong, and class variables persist until the class is removed from the library.
- Unbound objects are automatically collected and discarded by the garbage collector.
- A comment is a note inserted into a program for documentation.
- Smalltalk comments consist of text preceded and followed by the double quote character.
- When writing computer programs, clarity is usually the paramount consideration.

Exercises

In the first two exercises, write an algorithm and the corresponding code on a piece of paper, execute the task by clicking buttons in the *Pen world*, and check whether the generated code is the same as yours.

1. Draw letter S in the center of the drawing area of *Pen world 4*. The letter should be red, 50 pixels high, and 20 pixels wide.
2. Draw letter X in the center of the drawing area of *Pen world 4*. The letter should be green, 50 pixels high, and 20 pixels wide.
3. Try the more advanced *Farm* worlds. They use similar formats as *Pen* worlds.

3.3 Writing and executing programs

Up to this point, we have been 'programming' with buttons. In this section, we will switch to real programming using *Pen world 5* (Figure 3.5) which provides the same functionality as regular Smalltalk programming tools such as the Workspace.

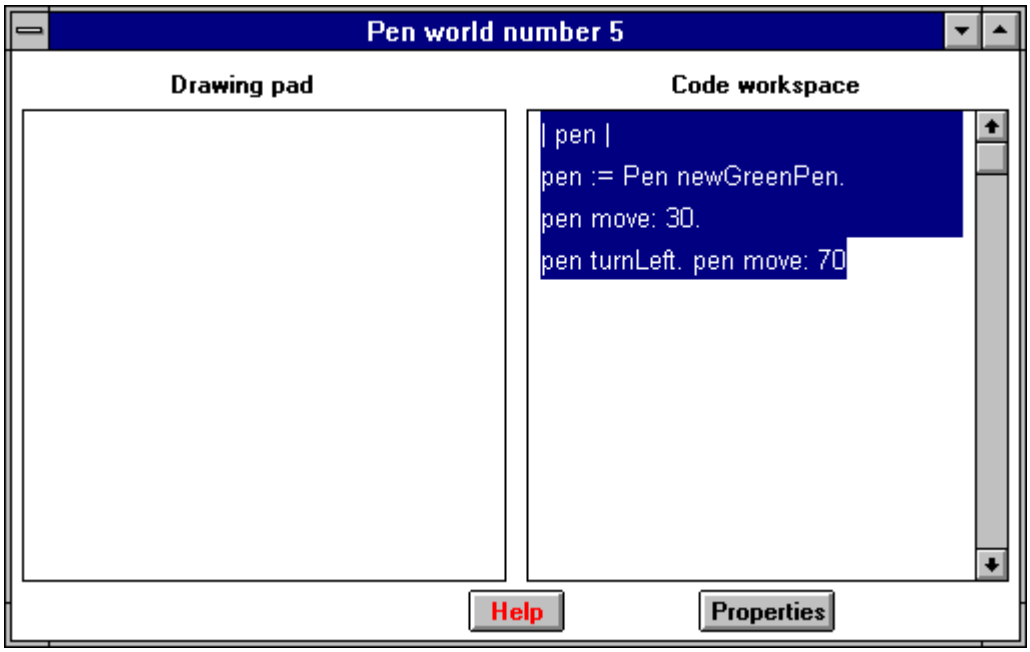

Figure 3.5. *Pen world* 5 with code selected for execution.

To write and execute a code fragment in *Pen world* 5, type the code into the text view, select it, and execute it with the *do it* command from the <operate> menu[3]. We leave it to you to write and execute some of the exercises at the end of this section. In all cases, write the code on a piece of paper first, enter it very carefully into the text view, and try to execute it. If you have problems, read the rest of this section to see how to recover. If you can't solve the problem, go to *Pen world 4*, execute the task using buttons, study the automatically created code, correct your program in *Pen world 5* accordingly, and try again.

[3] Since we have introduced two buttons so far - the <*select*> and the <operate> buttons, let's introduce the third one: If you have a three-button mouse, Smalltalk's name for the right mouse button is <*window*> *button* because it opens the <window> menu with window-related commands. If your mouse has only two buttons, pressing the <Ctrl> key and the right button has the same effect as pressing the rightmost key on a three-button mouse. In the first implementations of Smalltalk, the left, middle, and right buttons were called *red*, *yellow*, and *blue* and some important Smalltalk code still uses these names.

Dealing with simpler errors

Sooner or later, you will type and attempt to execute incorrect code. Code can be incorrect in more than one way:

- It may violate some rules of the language - the *syntax* rules[4]. As an example, you may have used illegal characters in an identifier name or typed a space between : and = in the assignment. If you enter syntactically incorrect code, the compiler will notify you and Smalltalk will not even start executing the code. Once you master the simple Smalltalk syntax, correcting syntax errors is easy.
- A syntactically correct code may use a meaningless identifier such as a misspelled or undeclared variable or class name, or a message name whose declaration does not exist. The compiler will again notify you and correcting the mistake should be easy, especially because the corrector can help you find correct spelling and the environment can declare variables for you.
- A frequent cause of errors is sending a legal message to the wrong object. As an example, even though the following expression contains a legal receiver and a legal message,

  'abc' squared

  is illegal because string objects don't understand the squared message (you cannot square a string.) This kind of error is usually easy to correct. When you try to execute code with this problem, Smalltalk will execute it up to the point where it breaks and open an Exception window. The use of Exception windows is explained below.
- The really unpleasant error is a *logic error*. In this case, everything is legal and executes but it does not produce the correct result. As an example, if you are trying to draw a square and turn the pen left instead of right, you obtain a drawing but not the one you wanted - this is an error in the logic of the program. If you have a logic error (your program works but produces an incorrect result), read your code very carefully and try to correct it. If this does not help, use the Debugger as explained below.

The first two kinds of errors listed above occur at *compilation time* (when the code is translated for execution and before it can be executed), the last two occur at *run time* (when encountered during the execution of the code). We will now illustrate some of the possible problems in more detail.

Misspelled name

If you misspell the name of a variable or a message, Smalltalk opens a Notifier window as in Figure 3.6. In this case, we incorrectly typed the name of a message as newredpen instead of newRedPen. In a minor misspelling such as this one, Smalltalk will be able to find the correct name if you click *correct it*. If the mistake is bigger, *correct it* will not be able to help. In this case, select *abort*, return to your program, and correct it manually.

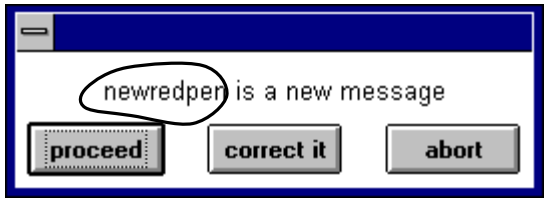

Figure 3.6. Error window indicating that code contains an unrecognizable name (here newredpen).

[4] See Appendix 7 for a complete listing of Smalltalk syntax.

Receiver does not understand message

If everything is typed correctly but the receiver does not understand the message because its class does not define it and does not inherit it, Smalltalk opens an Exception window such as the one in Figure 3.7. This Exception window popped up when we tried to execute

```
'75' squared
```

The message displayed in the window clearly identifies the problem (if you know that ByteString refers to strings) and we can thus close the Exception window by clicking *Terminate*, correct the expression to

```
75 squared
```

and re-execute the code with better success. If the problem is not clear, click *Debug* and open the Debugger. We will talk about the Debugger shortly.

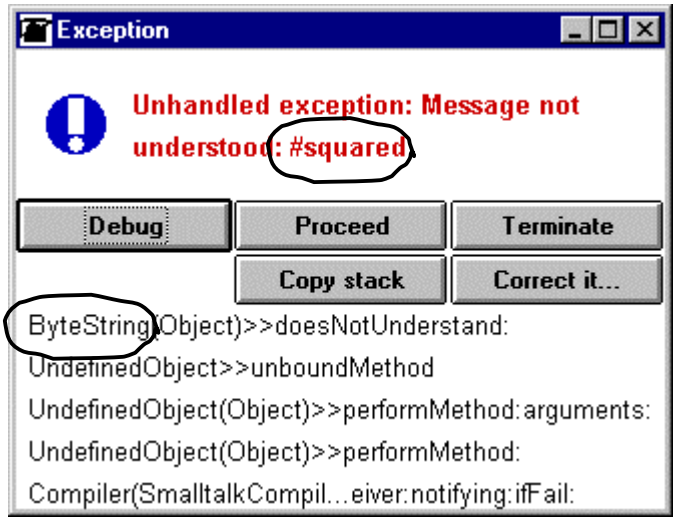

Figure 3.7. Exception window indicating that the receiver (a ByteString object) does not understand message squared. Click *Terminate*, return to your program, and correct the error.

Main lessons learned:

- On a three-button mouse, the right button is the <window> button. Use it to activate the <window> menu with window-related commands. If you only have two buttons, the right button in combination with <Ctrl> has the same effect.
- The <select>, <operate>, and <window> buttons were originally called *red*, *yellow*, and *blue* and some Smalltalk methods still use these names.
- The rules of a language are called syntax rules, and code that violates a syntax rule contains a syntax error.
- Besides syntax errors, code can contain run time errors such as messages sent to receivers that don't understand them, and logic errors.
- Smalltalk catches all but logic errors and opens notifier or exception windows explaining their nature.

Exercises

1. Write an algorithm and a program to draw two squares centered at the home position. The inner square should be red with size 40 pixels, the outer square should be blue with a side of 60 pixels.

2. Objects in *Pen World 5* can execute messages that we have not listed so far. As an example, you can change the width of the tip of the pen, its drawing speed and its color, and you can perform additional operations with the eraser as well. Use the System Browser to find, list, and explain all pen and eraser messages, formulate several problems using them, program them, and test them.
3. Open a Workspace and enter and execute the following code fragment. Be careful to type it correctly.

```
| number string |
"Get a number from the user."
string := Dialog request: 'Enter a number' initialAnswer: '0'.
number := string asNumber.
"Calculate and display its square in the Transcript."
Transcript cr.
Transcript show: 'The square of '.
Transcript show: number printString.
Transcript show: ' is '.
Transcript show: number squared printString
```

The program requests a number from the user, calculates its square, converts it to a string, and displays it in the Transcript. Identify all parts including the declaration, variables, assignments, receivers, messages, statements, and comments.

3.4 More about variables

We have, by now, encountered the term *variable* in several contexts and it is time to summarize and refine our definitions:

- A variable is an identifier bound to an object. Sending messages to a variable is equivalent to sending messages to the object referenced by the variable.
- Classes keep their properties in *class variables*. Class variables are listed in the definition of the class and their value is shared by the class and its subclasses. They can be accessed by the class, its subclasses, *and their instances*. The lifetime of objects bound to class variables is the same as the lifetime of the class and since classes are persistent (saved in the 'image' file[5] when you execute the *save as* or *exit* with *save and exit* command from the launcher window – and restored when you start Smalltalk), they last until the class defining them is deleted from the library.
- Instances keep their values in *instance variables*. Instance variables are private to the instance which means that they cannot be directly accessed by any other object. In particular, instance variables cannot be accessed by the class. Other objects can access an instance variable only if the class definition provides a method that accesses it. You will find all instance variables in the definition of the class and the lifetime of objects bound to instance variables is related to the lifetime of the instance: When an object ceases to exist, the objects bound to its instance variables are discarded and collected by the garbage collector *unless* they are also bound to some other variables or objects that are still 'live'.
- Variables declared in methods are called *temporary variables*. They are valid only within the method and their validity is limited to the method. When the execution of a method ends, objects bound to its temporary variables are discarded unless they are bound to some surviving objects.
- When a temporary variable is first encountered in a program or when an instance variable becomes valid when n instance is create, it is automatically bound to nil and new values can be assigned to it with the assignment expression. This assignment may happen only in the appropriate context:
    - Class variables may only be assigned in class or instance methods.
    - Instance variables may only be assigned in instance methods.
    - Temporary variables may only be assigned in the method in which they are declared.

[5] The image file is one of the essential components of the Smalltalk environment. See Appendix 8 for more about it.

The range of code in which an identifier is valid is called its *scope*. The scope of a temporary variable is thus the method or code fragment in which it is declared. The scope of an instance variable consists of the instance methods in the class in which the variable is declared and in all its subclasses. The scope of a class variable consists of the class-, subclass-, and instance methods.

Type of variable	Accessible from (scope)
temporary	method where declared
instance	instance method of class where defined and its subclasses
class	class and instance method of class where defined and its subclasses

To illustrate the concepts of instance and class variables, Figure 3.8 shows the definition of class Date. It shows that each instance of Date has two instance variables, one called day and one called year. The class also has five class variables and inherits several variables from superclasses. (The easiest way to see all class variables defined for a class is to use the *class var refs* in the <operate> menu of the class list or use the Full Browser in Advanced Tools.) Date does not inherit any instance variables. (Use *inst var refs* in the <operate> menu.).

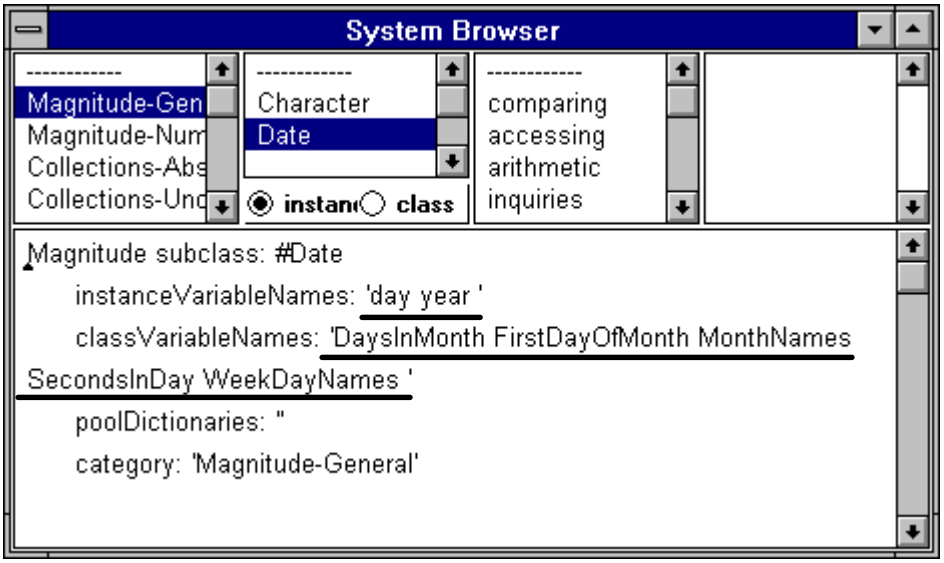

Figure 3.8. Definition of class and instance variables in class Date.

Inspecting variables

To see the components of an object, send it the message inspect. This will open an *Inspector* window. As an example, when you enter

```
(Date today) inspect
```

into a Workspace and execute it with *do it*, you will get an Inspector on the object returned by Date today (Figure 3.9). We used the parentheses only to indicate the logic of the expression (we are inspecting the object Date today) but we can get the same result by executing

```
Date today insect
```

or simply by executing

```
Date today
```

with the *inspect* command instead of using the inspect message and *do it*.

The self item at the top of the list refers to the receiver of the inspect message, in this case the Date object. The remaining items provide access to all instance variables of the inspected object, both inherited and defined in its class.

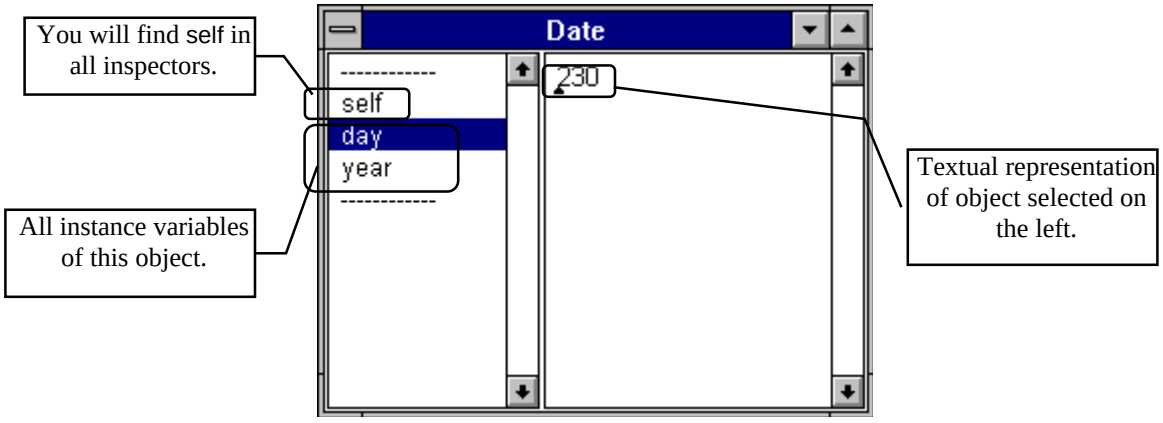

Figure 3.9. Inspector on the Date object created by Date today.

Class Date provides a good example of the need for class variables (Figure 3.10). Its class variable FirstDayOfMonth contains the number of the first day of each month in a year counted from the start of the year, MonthNames contains the names of all months, and so on. These objects are of interest to many class and instance methods in Date and this programs and is why they are stored in class variables

Besides viewing the values of an object's instance variables, the inspector can also be used to inspect the variables themselves. As an example, if you selected day as in Figure 3.9, executing *inspect* in the inspector's <operate> menu will open an inspector on the day object, and so on. Inspecting objects to any desired depth is one of the most common activities during program testing.

Since classes are also objects, we can inspect them too. To inspect *class* Date, enter

```
Date inspect
```

into a Workspace and execute it with *do it*, or execute *inspect* over the name of the class. This will open the inspector in Figure 3.10. Most of the items in the list don't make much sense at this point because classes are rather special objects, but classPool contains all class variables with their values. Another interesting item is variable methodDict which contains the list of all instance methods defined in this class.

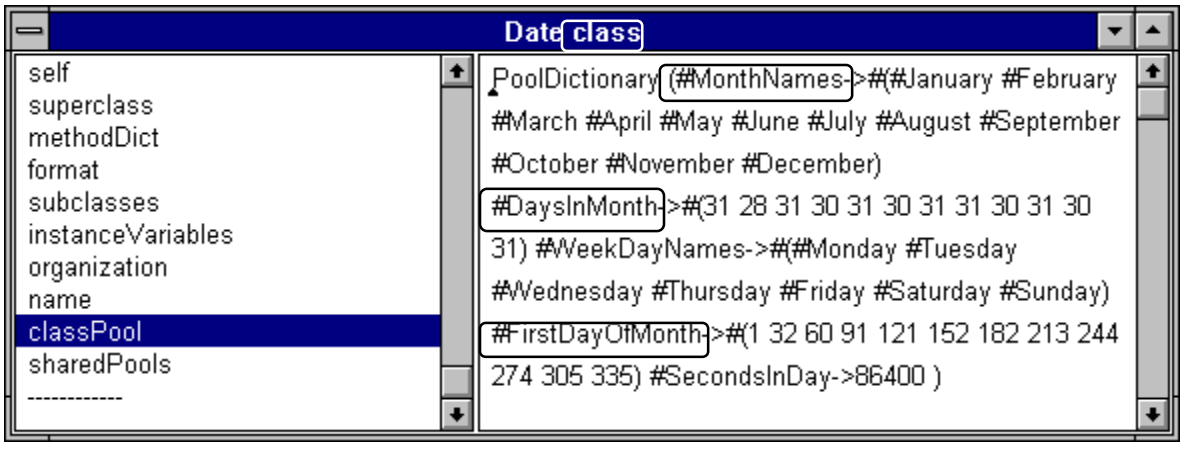

Figure 3.10. Inspector of class Date. The window label says that the inspected object is a class. Class variables are highlighted on the right side.

Main lessons learned:

- The scope of a variable is the range of code in which the variable can be directly used or assigned to.
- The scope of a temporary variable is the method in which it is defined, the scope of an instance variable are instance methods of the class and its subclasses, and the scope of a class variable are class and instance methods of the class and its subclasses.
- The lifetime of an object is the time during which it is valid. Except for certain base objects, an object is live when other live objects refer to it.
- Values of instance variables of an object can be inspected by an Inspector.
- To view class variables, inspect the class object itself and then inspect its classPool variable.
- Smalltalk uses the special identifier self to refer to the receiver of a message.

Exercises

1. Create the following objects and inspect them and their components:
   a. An instance of Time obtained with message Time now.
   b. An instance of Rectangle obtained with message Rectangle origin: 12@13 corner: 150@300.
   c. A Rectangle obtained with Rectangle origin: 12@13 extent: 300@400.
2. Inspect the class variables of the following classes: Point, Rectangle, Random, Time, Float. Try to explain their meaning and justify the need for them. (Hint: Read also the class comment.)
3. Examine ten randomly selected classes using the System Browser and count how many of them have instance variables and how many have class variables. What do you conclude about the relative frequency of instance variables and class variables defined in a class?

3.5 Smalltalk messages

After examining object properties (class and instance variables) we will now turn to messages. As you have already seen, Smalltalk recognizes three types of messages - *unary*, *binary*, and *keyword* messages and each of them returns a single object as its result. We will now outline the rules for forming their names (*selectors*) and then give more detailed explanations accompanied by several examples.

- A *unary message* consists of a single identifier such as turnLeft. It does not have any arguments.
- Selectors of *binary messages* use special symbols such as + and @ and have exactly one argument. As an example, + is a binary message used as in 3 + 4, and @ is used to create Point objects as in 13@45.
- The selector of a *keyword message* consists of one or more keywords and each keyword is an identifier followed by a colon. Each keyword is followed by an argument as in move: 70. Note that the colon immediately follows the text without any intervening blanks.

All three types of messages may be used for either class or instance messages. The built-in library contains many unary, binary, and keyword methods and you can define your own messages of any type as well, thus creating your own 'vocabulary'. Smalltalk treats built-in and user-defined messages exactly the same way. We will now show some examples of all three kinds of messages and we encourage you to execute them and inspect their results.

Unary messages

We have already encountered several examples of unary messages including

Pen **newBluePen**	"A *class* message (receiver is *class*) Pen which creates a new pen."
CatFactory **new**	"A *class* message which creates a new cat."
cat **meow**	"An instance message (receiver is an *instance of class* Cat)."
pen **turnLeft**	"An instance message."

Time **now**	"A *class* message that returns a Time object for the current time."
cow **isHungry**	"An instance message."

The following are several more rather self-explanatory examples:

3.14 **negated**	"**negated** is an instance message that changes the sign of a number."
Random **new**	"**new** is a *class* message used by most classes to create a new instance."
100 **factorial**	"**factorial** is an instance message, here calculating 100*99*98*97* ... *3*2*1."
0.3 **cos**	"**cos** is an instance message that calculates cos(x), the cosine of its receiver."
25 **squared**	"**squared** returns the square of its receiver."
'Abc' **asUppercase**	"**asUppercase** converts its receiver to uppercase letters, here 'ABC'."
3.14 **asFraction**	"**asFraction** is an instance message that converts a number to a Fraction."

Binary messages

Binary messages are used mainly for arithmetic operations and for very common one-argument messages. A binary selector can be formed by one or more symbols[6] selected from a variety of characters including + - , / \ * & < > = and @. Several other characters such as ! and ? are also allowed but never used.

The following are several examples of binary messages. Type them into a Workspace and execute them with *inspect* or *print it* to see the result.

100 @ -340	"Message to 100 - creates a new Point object."
3 **+** 17	"This and the following six binary messages do arithmetic."
5 **-** 175	
56 ***** 43	
187 **/** 23	"This and the following three messages perform various division operations."
187 **//** 23	
187 **** 23	
187 **** 23	
'the name of the file is', ' ass.12'	"Combines (concatenates) two strings into one."
(12@15) **+** (30@40)	"Adds together two points - vector arithmetic."
3 > 8	"Returns the false object because it is not true that 3 is greater than 8."
3 <= 5	"Returns the true object because 3 is less or equal to 5."

All these messages are instance messages as their receivers are instances of numbers or strings. There are no binary class messages but there is no rule against defining one.

Keyword messages

Keyword messages are the essence of Smalltalk's expressive power because they let you create methods with any number of arguments. In fact, Smalltalk could exist without unary messages (one could use keyword messages and ignore arguments) and without binary messages (they could be implemented as keyword messages with one keyword). One could say that unary and binary messages are included just for convenience and readability.
The following are a few examples of keyword messages that you should try out:

Keyword messages with a *single keyword*:

Dialog **confirm:** 'Delete file?"	"Class message; opens dialog window, returns true or false."
Dialog **warn:** 'The disk is now full.'	"Opens a warning window with an OK button."
'abcdef' **isLike:** 'abc*'	"Returns true - checks *wild card character* * matches."
'aVeryLongFilename' **chopTo:** 8	"Returns 'aVername' - original chopped to 8 characters."

[6] The traditional rules allow only one or two special characters and impose special rules on the use of the – character. The current Smalltalk Standard Draft proposal removes most of these restrictions.

```
'a cat' spellAgainst: 'a car'          "Returns 80 - degree of similarity between the two strings."
```

Whereas one-keyword messages don't seem to cause any problems, keyword messages with multiple keywords may be more difficult to read. Beginners often don't understand that the multiple keywords form a single message. The following are a few examples of multiple-keyword messages, one message per line.

Keyword messages with *two keywords*:

```
Dialog request: 'Name' initialAnswer: 'John'     "Opens dialog, returns string entered by user."
Rectangle origin: 10@20 corner: 100@200          "Class message creating a rectangle."
3 between: 10 and: 20                            "Returns false: 3 is not between 10 and 20."
```

Keyword message with *three keywords*:

```
'Smalltalk' changeFrom: 1 to: 5 with: 'Big'      "Replaces the first five characters, giving Bigtalk."
```

Keyword messages with more keywords are allowed but are not very common.

In closing, let us note that the name of a message excluding arguments is called the *message selector*. As an example, between:and: is the selector of message between: 5 and 19; + is another selector and 3 + 5 is an expression using it.

Main lessons learned:

- Smalltalk recognizes unary, binary, and keyword messages.
- Unary messages consist of an identifier only and do not have an argument
- Binary messages are formed by combining one or two special symbols from a set of characters including + - / \ * & @ , < > =. A binary message has exactly one argument.
- Keyword messages are a sequence of one or more identifiers followed by a colon and an argument.
- A keyword is a selector followed by a colon.
- User-defined and built-in messages are treated in the same way.
- The name of a message (without its arguments) is called the message selector.

Exercises

1. Execute the code fragments from this section.
2. Examine 10 randomly selected method protocols using the Browser and estimate the relative frequency of unary, binary, and keyword messages.

3.6 Combining messages

Each message returns an object and since this object can be used as the receiver of another message or as an argument, messages can be combined or *chained*. As an example, if you want to add together two points, you can write either

```
| point point1 point2 |
point1 := 12 @ 15.
point2 := 37 @ -81.
point  := point1 + point2.
etc.
```

or simply

```
| point |
point  := (12 @ 15) + (37 @ -81).
etc.
```

if you don't need the intermediate points. If you do need the two intermediate points, you can also write

```
| point point1 point2 |
point  := (point1 := 12 @ 15) + (point2 := 37 @ -81) ).
etc.
```

because an assignment returns the value of its right hand side.

Execute all three forms with *print it* or *inspect* in a Workspace to see that you will get exactly the same result. As another example, to test whether 12^{15} /5 is less than 20 factorial, you could execute

```
| number1 number2 |
number1 := 12 raisedTo: 15.
number1 := number1 / 5.
number2 := 20 factorial.          "20*19*18*17*...*4*3*2*1."
number1 < number2                 "Returns true or false. "
```

or simply

```
((12 raisedTo: 15) / 5) < (20 factorial)
```

Execute both fragments to see that they both return the same result.

All experienced Smalltalk programmers combine messages to simplify code and to eliminate unnecessary variables. However, don't overuse this feature because expressions can become too complicated and difficult to understand, possibly producing different results than anticipated.

Obviously, if you want to combine messages and get the correct result, you must know how Smalltalk interprets combined messages. This will be explained in the next section.

Main lessons learned:

- Each Smalltalk message returns an object and this object can be used as an argument or as the receiver of another message. Combining (chaining, nesting) messages in this way makes code more compact.
- Chaining is common but use it with moderation to keep the code readable.

Exercises

1. Read the following expressions and classify them as unary, binary, and keyword messages. Try to predict their meaning and execute them.
   a. 3 + 2 factorial
   b. (3 + 2) factorial
   c. 0.3 sin squared + 0.3 cos squared
   d. (0.3 sin squared) + (0.3 cos squared)
   e. 15 factorial between: (15 raisedTo: 7) and: (15 raisedTo: 22)
   f. 12@15 corner: 123@75 "Creates a rectangle using two point objects."
   g. (Point x: 27 * 0.7 cos y: 45 * 0.7 sin) < 45@34 "Is Point on left above and to the right of the Point on the right?"
2. Imagine two solutions to the same problem. The first solution stores a large object in a variable, the second avoids the need for a variable by combining messages into larger expressions. Explain the possible advantage of the first approach.
3. What do you think is the effect of the following expressions. Test your prediction.
   a. (3 + 5) / 2 "For integer receiver and argument, message / returns a fraction."
   b. Rectangle origin: ((12@15) + (17@18)) extent: (90@x5 * 3))
   c. Dialog request: ('Enter name of ', string) initialAnswer: ('file.', extension)
      assuming that the value of variable string is 'the file', and that extension is '*'.
3. Explain the differences between the following expressions:
   a. x := 'abc' asUppercase
   b. (x := 'abc') asUppercase

c. y := (x := 'abc') asUppercase

4. Write Smalltalk expressions to solve the following problems and test that they calculate the correct result by typing your code into the Workspace and evaluating it with *print it*.
   a. The hypotenuse and one side of a right-angle triangle are 5 and 4 points long respectively. Calculate the third side. (Hint: There are two messages to calculate the square. One is **, the other is squared. The message to calculate the square root is sqrt.)
   b. What is the angle between the two given sides in the previous example? (Hint: Look in class Number for the required messages.)
   c. In a certain country, sales tax consists of a part that goes to the federal government and a part that goes to the provincial government. The first tax is 7 percent of the price, the second tax is 8 percent of the result of adding the first tax to the price. Write an expression to calculate the total amount payable for an item whose price is $20.
   d. A triangle has sides that are 45 and 30 units long. The angle between the two sides is 45 degrees. Calculate the length of the third side. (Hint: Trigonometric functions use radians rather than degrees. Use message degressToRadians to perform the conversion.)

3.7 Nesting of Expressions

Without knowing how Smalltalk interprets combined messages, we cannot know what a combined message calculates. Fortunately, the rules are very simple and usually correspond to the order in which expressions are evaluated. The formal definition of the rules constitutes the syntax of Smalltalk (Appendix 7) but since their interpretation by a beginner would not be easy, we give the following informal description of expression evaluation:

1. All expressions in *parentheses* are evaluated first, from left to right. This evaluation uses steps 1 to 4 again as necessary.
2. All *unary* messages are evaluated next, from left to right.
3. All *binary* messages are evaluated next, from left to right.
4. The remaining *keyword* message, if any, is evaluated.

To clarify these simple but rather abstract rules, we will now give several examples of Smalltalk expressions and evaluate them step by step. Each example adds a new feature to the discussion. Our detailed analysis may make execution appear complicated but this is because we are tracing the rules in a machine-like fashion. If you follow the rules in a more humane way, the execution will appear much simpler.

Example 1

Consider the following statement, presumably a part of a larger code fragment:

```
...
3 factorial raisedTo: 3.
...
```

This statement is executed as follows:

Step 1. *Parenthesized messages*: There are no parentheses so nothing happens in this step.
Step 2. *Unary messages*: factorial is a unary message. How do we know that? The first item in the statement - number 3 - must be the receiver because every expression begins with a receiver; consequently, factorial must be a message. It cannot be a binary message (these use special characters), and since it is not followed by a colon, it is not a keyword message. So it must be a unary message.
As a consequence, 3 factorial is evaluated first, giving 3*2*1 = 6 and the original expression can now be thought of as

```
6 raisedTo: 3
```

There are no more unary messages in this statement so we are finished with Step 2.
Step 3. *Binary messages*: None in this statement (no special symbols).
Step 4. *Keyword message*: Message raisedTo: is a keyword message (it follows a receiver and consists of an identifier followed by a colon) with argument 3. Executing it returns $6^3 = 216$.

Note that the expression evaluates exactly as

```
(3 factorial) raisedTo: 3
```

Example 2

The following statement attempts to test whether 12^4 is between 7^5 and 10^5 and store the result in temporary variable condition:

```
| condition ... |
...
condition := 12 raisedTo: 4 between: 7 raisedTo: 5 and: 10 raisedTo: 5
...
```

Unfortunately, this statement will not even start executing because we did not write it properly. When you examine what we wrote, you will see that 12 is the receiver, so raisedTo: must be a keyword. Number 4 is its argument and between: is another keyword. Continuing in this way, we find that we are telling Smalltalk to execute a keyword message with selector raisedTo:between:raisedTo:and:raisedTo:. Such a message is not in the library and Smalltalk will not accept the code. This is an example of a statement that is syntactically correct but invalid it because its interpretation implies a non-existent message. Smalltalk will catch this error and display the dialog in Figure 3.11.

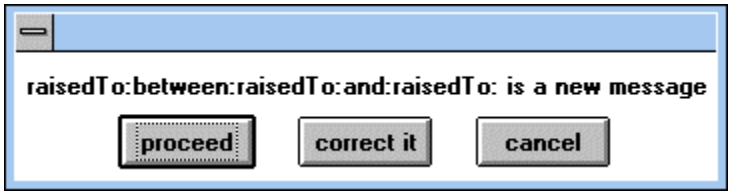

Figure 3.11. Error produced when trying to execute the first version of Example 2.

Note that the result of sloppy writing could be much worse: If we wrote the expression in such a way that its interpretation does give a message which is in the library, Smalltalk would execute it even if the message was different from the one that we had in mind. This shows that it is important to be very careful when combining messages and to make sure that the code is easily readable. We will now rewrite the assignment to produce the desired effect.

In essence, we are trying to determine whether a certain number is between two other numbers. We will thus put parentheses around the expressions that calculate the three numbers and use them as the receiver and the two arguments of between:and: respectively as follows:

```
| condition ...|
...
condition := (12 raisedTo: 4) between: (7 raisedTo: 5) and: (10 raisedTo: 5)
...
```

Evaluation of this statement proceeds as follows:

1. *Parenthesized expressions* from left to right. Parenthesized expression

```
12 raisedTo: 4
```

is evaluated by re-applying evaluation rules:

1. *Parentheses*: Expression 12 raisedTo: 4 does not contain any parenthesized expressions.
2. *Unary*: It does not contain any unary messages.
3. *Binary*: It does not contain any binary messages.
4. It contains the *keyword* message raisedTo: and when we evaluate it, we get 20736. This reduces the original expression to

```
20736 between: (7 raisedTo: 5) and: (10 raisedTo: 5)
```

We are still in Step 1 of the original evaluation and we thus proceed scanning the original statement looking for more parenthesized expressions. We now find the parenthesized expression

```
(7 raisedTo: 5)
```

and evaluate it just as the previous parenthesized expressions, getting 16806. The original expression has now been reduced to

```
20736 between: 16806 and: (10 raisedTo: 5)
```

We are still not finished with Step 1, and proceeding to the right we find the last parenthesized expression

```
(10 raisedTo: 5)
```

evaluate it, and obtain 100000. The original statement now effectively reduces to

```
condition := 20736 between: 16806 and: 100000
```

2. This form of the original expression does not contain any *unary* messages.
3. There are no *binary* messages.
4. The expression contains the *keyword* message between:and: with receiver 20736 and arguments 16806 and 100000. Evaluation returns object true because it is true that the value of the receiver is between the values of the two arguments.
5. Object true, the result of the last step, is assigned to variable condition and execution proceeds to the next statement in the program.

Example 3

Using evaluation rules, you can find that

```
3 + 7 * 3
```

is evaluated as

```
(3 + 7) * 3 = 10 * 3 = 30
```

because Smalltalk evaluates from left to right with no exceptions. In conventional arithmetic, multiplication has precedence over addition and the result would be the same as for

```
3 + (7 * 3) = 24
```

If this is what you intended, you must use parentheses as in

(3 + 7) * 3

Parenthesizing arithmetic expressions is highly advisable even when it is not required because misinterpreting arithmetic expressions is a very frequent error in Smalltalk programs. Don't worry about making your programs less efficient by using unnecessary parentheses – the compiler will ignore unnecessary parentheses when it creates the executable code.

Example 4

In a fictitious program to calculate tax using notoriously obscure tax rules, we need to add the sum of variables income1 and income2 multiplied by the weighting factor 0.13 to income3 multiplied by 0.17, and subtract the sum of variables expenditure1 and expenditure2 weighted by 0.06. The result is stored in variable tax.

One way to solve this problem is to declare a variable for each component of the calculation and then do the final calculation on these variables as follows:

```
| part1 part2 part3 income1 income2 income3 expenditure1 expenditure2 ...|
income1 := 1000.
income2 := 2000.
income3 := 100.
expenditure1 := 500.
expenditure2: = 700.
part1 := (income1 + income2) * 0.13.
part2 := income3 * 0.17.
part3 := (expenditure1 + expenditure2) * 0.06.
tax := part1 + part2 - part3.
...
```

This program directly reflects the original formulation and we are thus quite confident that it works correctly. However, since the calculation is not all that complicated, we decide to eliminate the auxiliary variables and parenthesize each part of the statement to capture the calculations described in the specification. In a way, the parentheses replace the auxiliary variables which we don't need in the long term anyway. This gives the following formulation:

```
| income1 income2 income3 expenditure1 expenditure2 ...|
income1 := 1000.
income2 := 2000.
income3 := 100.
expenditure1 := 500.
expenditure2: = 700.
tax := ((income1 + income2) * 0.13) + (income3 * 0.17) - ((expenditure1 + expenditure2) * 0.06).
...
```

Evaluation of the last statement proceeds as follows:

1. *Parenthesized* expressions from left to right. The first parenthesized expression is

   (income1 + income2) * 0.13

   because we read the parentheses from the outside, and to evaluate it we proceed as follows:
   1. Evaluate the first parenthesized expression

      income1 + income2

      by applying the five steps again:

1. No *parenthesized* expressions.
2. No *unary* messages.
3. We have *binary* message +. Smalltalk sends + to the object referenced by income1 and checks gets the result 3000. Steps 4 and 5 don't apply and we return to the original level of evaluation.

Expression

```
(income1 + income2) * 0.13
```

has now been reduced to 3000 * 0.13.
There are no more parenthesized expressions at this level and we continue to the next rule.
2. There are no unary *messages* in 3000 * 0.13.
3. Expression 3000 * 0.13 contains the *binary* message * and evaluates to 390. This ends the evaluation of this part of the statement (no keyword messages, no assignment) and we return to the original statement which is now effectively reduced to

```
tax := 390  + (income3 * 0.17) - ((expenditure1 + expenditure2) * 0.06).
```

We leave it to you to evaluate the rest of the statement and to check that the result is the same as the result of the formulation using temporary variables. To do this, execute and inspect both versions in the Workspace.

Main lessons learned:

- Smalltalk uses the following rules to evaluate expressions:
  1. Evaluate all parenthesized expressions from left to right applying rules 1 to 4 to their contents.
  2. Evaluate all unary messages from left to right.
  3. Evaluate all binary messages from left to right.
  4. Evaluate the resulting keyword message, if any.
- In applying evaluation rules, Smalltalk does not make any exceptions. In particular, it does not give precedence to any arithmetic binary messages.
- Combining expressions makes code more compact and eliminates variables. It may, however, make code difficult to understand and should not be overused.
- If an expression becomes difficult to read, divide it into consecutive statements and use temporary variables or parenthesize it.
- Be very careful with arithmetic expressions - conventional arithmetic operators use precedence but Smalltalk does not.
- Be careful with keyword message, especially when their arguments are calculated. They can be hard to understand and write correctly.

Exercises

1. Check the correctness of the syntax of the following expressions, identify receivers, arguments, and messages, and convert them to more readable but equivalent forms where appropriate. Interpret their meaning and execute them where possible.
   a. 3 + 4 * x // 37 - 5
   b. self account add: 1.4 * bill to: (transactions invoiceFor: currentTransaction)
   c. (5 raisedTo: 3 + 7*x) gcd: 5 factorial + x
2. Explain the order of evaluation of the following expressions and represent it by bracketing. Execute them with *print it* to confirm your analysis.
   a. 3 + 2 factorial
   b. (3 + 2) factorial
   c. 0.3 sin squared + 0.3 cos squared
   d. (0.3 sin squared) + (0.3 cos squared)
   e. 15 factorial between: (15 raisedTo: 7) and: (15 raisedTo: 22)

f. 12@15 corner: 123@75
g. (Point x: 27 * 0.7 cos y: 45 * 0.7 sin) < 45@34

5. Evaluate the following expressions and test the correctness of your analysis by evaluating the expressions in a Workspace.
   a. (3 + 4) factorial / 2 squared
   b. 0.7 sin between: 0.4 sin and: (0.1 cos) squared
   c. (Rectangle origin: (10@30) corner: (100@70)) intersects: (Rectangle origin: (75@97) extent: (30@40))
6. Can any of the parentheses in the above exercises can be deleted without affecting the result?
7. Examine 20 randomly selected method definitions and find the
   a. average number of lines in a method.
   b. average number of statements per method.
   c. percentage of statements using combined messages.
   d. average number of messages per statement.
8. Check what happens when we modify the last statement of the tax example as follows:

```
| income1 income2 income3 expenditure1 expenditure2 ...|
income1 := 1000.
income2 := 2000.
income3 := 100.
expenditure1 := 500.
expenditure2: = 700.
tax := income1 + income2 * 0.13 + income3 * 0.17 - expenditure1 + expenditure2 * 0.06.
```

   The syntax of the last statement is correct, the message sends are all legal because the messages are understood by their receiver, but the logic implied by the order of evaluation is wrong and so the result is also wrong.

3.8 Tracing message evaluation with the Debugger

In the previous section, we introduced message evaluation rules and demonstrated them by manual evaluation. We will now show how you can use the Debugger to do this for you.

Example

We want to execute the following code fragment step-by-step using the Debugger:

```
| x |
x := 7.
x := 3 factorial raisedTo: 3 + x.
```

To trace execution with the Debugger, we will add a halt message at the point where we want to start tracing, in our case just before the first assignment:

```
| x |
self halt.                    "Open an Exception window to allow continuation in the Debugger."
x := 7.
x := 3 factorial raisedTo: 3 + x
```

The halt message opens an Exception window which allows you to open the Debugger to trace the execution of the program and inspect its components. Since halt breaks the execution of the program, inserting a halt message is usually referred to as *setting a breakpoint.* The halt message is defined in class Object and it is thus inherited by every class and understood by all objects. Smalltalk programmers usually send halt to self - the receiver of the message that is currently executing. A code fragment is treated as an unboundMethod message sent to nil, and self in a code fragment thus refers to nil.

Enter the program into a Workspace and execute it with *do it*. When execution starts, it immediately opens the Exception window in Figure 3.12. The five lines of text below the buttons are the

last five currently active messages with the latest message halt on the top. This part of the display is referred to as the top of the message stack or *call stack*. To continue, click *Debug* to open the Debugger.

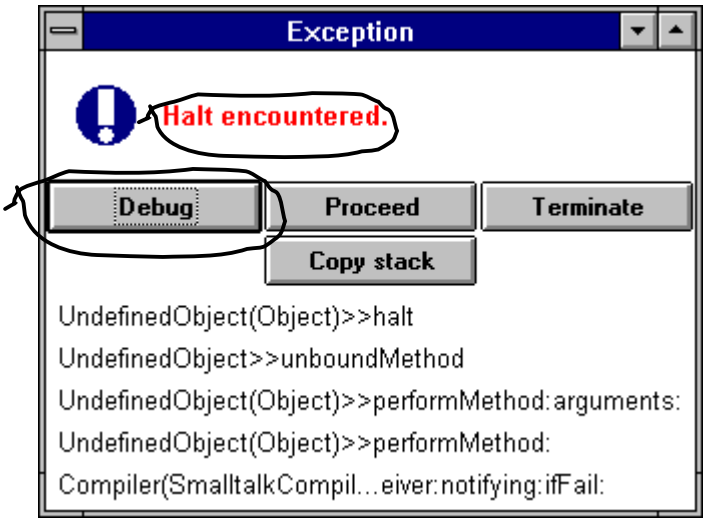

Figure 3.12. Exception window resulting from the execution of a halt message .

Clicking *Debug* opens the Debugger window in Figure 3.13. The scrollable top view of the window provides access to the call stack, the middle view is for displaying the code of a method selected from the stack, and the bottom part contains two inspectors. We will now explain these parts and illustrate their use.

As we already mentioned, the top message in the call stack is the method currently being executed, the one below is the method that sent the message on the first line, and so on. Each line consists of the name of the class of the object that received the message followed the name of the message. If the message is inherited, the class that defines it is given in parentheses. In our case, the currently active message is halt, it was sent by an UndefinedObject, but its definition is in the Object superclass of UndefinedObject.

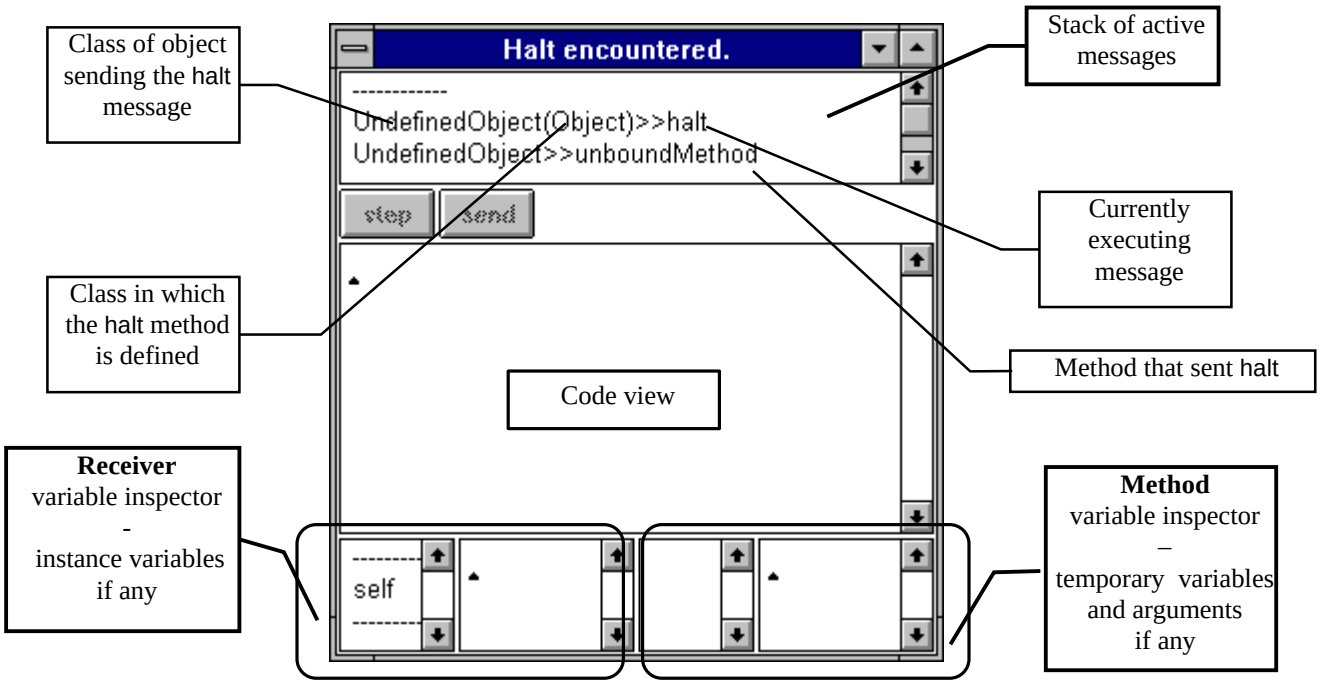

Figure 3.13. Debugger window with two most recently invoked methods shown on top.

To see the code of a message and the point reached in its execution, click the appropriate line in the call stack. Since a code fragment is always referred to as unboundMethod, we click the second line from the top and the Debugger changes to Figure 3.14. The message currently being executed is highlighted and the inspector at bottom right provides access to all temporary variables, arguments (none), and instance variables (none).

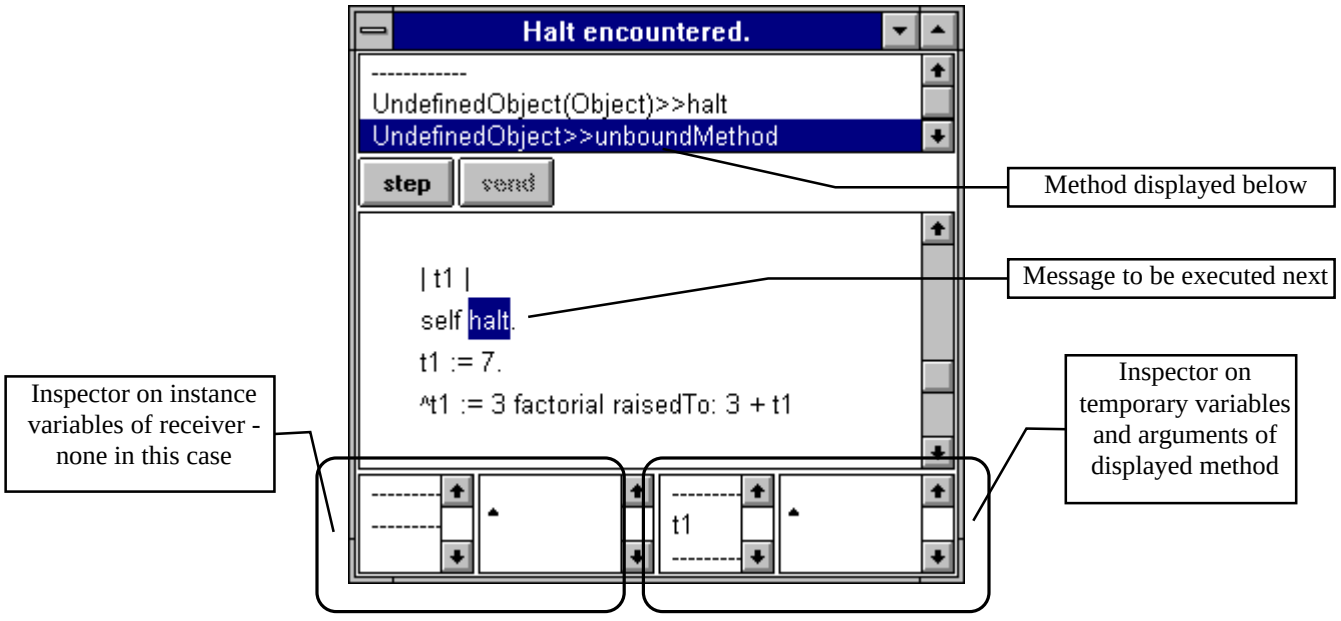

Figure 3.14. Debugger showing executing method and highlighting its current state of execution.

As you can see, the code is exactly like our code fragment except for variable names. This is because when the Debugger gets the message to display the code fragment, it reconstructs the code from a

condensed representation which does not include variable names, and it thus creates its own variable identifiers such as t1 and t2. The process of converting compressed representation back to the fully expanded *source code* is called *decompilation* because it is the reverse of compilation and the comment at the beginning of the code tells you that this is decompiled code. Decompilation always happens with code fragments so you can ignore the warning. It should not, however, happen with methods from the library; if it does, the directory path to your Smalltalk files are incorrectly set and must be changed on the *Sources* page of the *Settings* command under *Files* in the *Launcher* window.

Another difference between the original program and the code in the Debugger is the *circumflex* symbol ^ (called the *return* symbol in Smalltalk) in front of variable t1 on the last line. A caret in a method tells Smalltalk to stop executing the method at this point and return the object obtained by the expression following it, in this case the value of t1. In our example, the circumflex symbol was inserted by the compiler. We will ignore its use for now.

Back in the Debugger, we now have several options what to do next: we can continue execution within the Debugger, close the Debugger and continue execution of the program, or close the Debugger and terminate execution.

If we choose to *continue execution within the Debugger*, we can either click the *step* button or the *send* button: If we click *step*, the Debugger will execute the highlighted expression, update the inspectors, and proceed to the next message. If we click *send*, the Debugger will go inside the definition of the message being sent and execute it under your control. In other words, *send* allows you to get deeper and deeper into the execution of the code whereas *step* keeps you at the current level.

The remaining main options are *to close the Debugger window* (this will terminate the execution of the current message and the whole program that sent it), or to leave the Debugger and proceed with the execution of the program. To *proceed*, select *proceed* in the <operate> menu of the stack view.

In our case, we want to continue execution in the Debugger but we don't want to get any deeper at this point and we will thus click *step*. First, however, select variable t1 in the bottom right inspector so that you can observe how its value evolves as we execute the program. (t1 in the Debugger corresponds to x in our program). The initial value of t1 is nil because we have not assigned any value to it yet.

Click *step* a few times to execute a few steps until you reach the first assignment which assigns 7 to x and note the change in the inspector. The second assignment now begins execution by evaluating the expression on the right hand side. Smalltalk first checks for parenthesized expressions, does not find any, and looks for unary messages. It finds factorial, gets ready to execute it, and so on.

We leave the rest to you. Note, however, that the Smalltalk compiler sometimes generates code that does not evaluate in the order described by our four rules. This, however, does not change anything on the validity of our interpretation – although the code may be evaluated in a slightly different order, it will still produce the same result.

Other important properties of the Debugger and the Inspector

The code view in the middle of the Debugger can execute commands such as *do it*, *inspect*, and *print it* on any part of the code in the window. The Debugger also lets you change the code, recompile it with *accept*, and restart from the beginning of the method. This feature is frequently used because one often discovers mistakes while debugging, and on-the-spot modification and continued debugging allows you to test that a correction works. Note, however, that if the execution of the original code changed the values of some variables, execution will continue with these new values - unless you reset them using the inspectors at the bottom. To change the value of a variable or an argument, select it in the appropriate inspector, type the new value in the value part of the inspector window, and *accept* it from the <operate> menu>. Note that you can also do this by evaluating an expression typed into the inspector view.

Main lessons learned:

- The Debugger has been designed for finding and correcting errors in code but it can also be used for step-by-step execution at various levels of detail.
- The Debugger can be closed at any point, and execution terminated or continued in normal way.

- The top window in the Debugger shows the call stack, a list of currently active messages with the one currently executing on the top.
- The code view of the Debugger has all code view properties found in the Workspace.
- Inspectors in the Debugger window show values of instance variables, method arguments, and temporary variables. All can be changed during debugging.
- The code displayed by the Debugger can be edited and recompiled, and its execution resumed from the start of the method.

Exercises

1. Modify the following code fragments to open the Debugger just before the last statement and trace its execution step-by-step. What is the result of the last statement?
   a.
   ```
   |x y|
   x := 17.
   y := (Dialog request: 'Enter a number' initialAnswer: '') asNumber.
   (y+ 4) factorial / 2*x squared
   ```
   b.
   ```
   |arg1 arg2|
   arg1 := (Dialog request: 'Enter the sin argument initialAnswer: '') asNumber.
   arg2 := (Dialog request: 'Enter the cos argument initialAnswer: '') asNumber.
   arg1 sin between: 0.4 sin and: arg2 cos squared
   ```
2. Repeat Exercise 1.a and when the Debugger window opens, modify the 'Enter a number' part of the third line of the code to read 'Enter the value of y'. *Accept* the new text and continue. Comment on the result.
3. Repeat Exercise 1.b and when the Debugger window opens, change the value of variables arg1 and arg2 using the Debugger's inspector and continue.
4. Repeat Exercise 1 and when the Debugger window opens, re-execute the last Dialog message within the Debugger by *inspect*. Comment on the result.
5. Execute expression Date today in the Debugger. Use the inspector to examine the instance variables of the class object Date (they are the *class* variables from the definition of class Date).
6. Repeat the previous exercise with expression Time now.

3.9 Cascading

In this section, we will explain cascading, a Smalltalk feature that saves typing when a sequence of messages is sent to the same receiver. We will start by demonstrating why it may be useful, using an example that will also show how to output intermediate results.

In the previous section, we used the Debugger to monitor the execution of code fragments by adding breakpoints. Another way to track program execution is to let the program write intermediate results to the Transcript window, the bottom area of the launcher. VisualWorks itself uses the Transcript to display information about some aspects of its internal operation and since the Transcript is easily accessible from any program, Smalltalk programmers often use it for debugging or for intermediate output, before creating a proper user interface for an application.

To write to the Transcript, send messages to the Transcript object. The most important messages that it understands are

clear	- to clear (erase) the Transcript view
cr	- to start a new line in the Transcript view (cr stands for carriage return)
tab	- to insert a tab in the text
show: aString	- to print aString in the Transcript view

As an example of the use of show:

```
Transcript show: 'This is a test of the show: message'
```

writes the string enclosed between apostrophes to the Transcript.

The show: message expects a string argument and if you want to print information about an object that is not a string - for example a number - you must first convert first it to a string. This is achieved by message printString which is understood by all Smalltalk objects because it is defined in class Object. As an example, to print number 37 in the Transcript, use

```
Transcript show: 37 printString
```

where 37 printString converts number 37 to '37' which is a string. Message printString is also used by the *print it* command and by the inspector (to display the self object) and you are thus familiar with the kind of output that it produces.

As another example, we will now show the use of the Transcript for intermediate results. The following code fragment is a simple tax calculation that requests data from the user and records intermediate results in the Transcript:

```
| income expenses tax |
"Request income and expenses."
income := Dialog request: 'Enter income' initialAnswer: ''.        "Reads a string."
income := income asNumber.                                        "Converts the string to a number."
expenses:= Dialog request: 'Enter expenses' initialAnswer: ''.
expenses:= expenses asNumber.
"Calculate tax."
tax := (income - expenses) * 0.17.
"Display input and results in Transcript."
Transcript clear.
Transcript show: 'Income '.
Transcript show: income printString.
Transcript cr.
Transcript show: 'Expenses '.
Transcript show: expenses printString.
Transcript cr.
Transcript show: 'Tax '.
Transcript show: tax printString
```

The output produced in the Transcript window for income 20,000 and expenditure 4,700 is as follows:

```
Income 20000
Expenses 4700
Tax 2601.0
```

Enter the code fragment into a Workspace and execute it to see how it works. Note that the comma separating thousands should not be typed as input.

Save typing by cascading messages

The tax program that we have just written ends with a long sequence of messages to the Transcript. To reduce the clutter and to minimize typing, use *cascading*: When the same object is the receiver of a sequence of consecutive messages, you don't have to repeat it. Instead, replace the period at the end of the statement with a semicolon and delete the repeated receiver. With cascading, our original program can be written much more compactly as

```
| income expenses tax |
"Request income and expenses."
income := Dialog request: 'Enter income' initialAnswer: ''.
income := income asNumber.
expenses:= Dialog request: 'Enter expenses' initialAnswer: ''.
expenses:= expenses asNumber.
"Calculate tax."
tax := (income - expenses) * 0.17.
```

```
"Display input and results in Transcript."
Transcript clear;
        show: 'Income ';
        show: income printString;
        cr;
        show: Expenses ';
        show: expenses printString;
        cr;
        show: 'Tax ';
        show: tax printString
```

We used indentation to make it quite obvious that all the cascaded messages are sent to Transcript. Smalltalk programmers use cascading often because it saves typing and can make programs more readable. One place where cascading is used very often is in creating and initializing objects. As an example, a method creating a student object might contain the following cascaded statement:

```
self new firstName: firstName; lastName: lastName; id: aNumber
```

All the cascaded messages are sent to the object resulting from self new which shows that a sequence of cascaded messages can be sent to the result of an expression. Some programmers misunderstand how cascading works in this situation and either write programs that don't work correctly or don't use cascading. The precise rules for cascading are as follows:

- Cascaded expressions are separated by semicolons.
- All cascaded message have the same receiver as the last message *executed* before the first semicolon.

In our last example, the last message executed before the first semicolon is firstName: firstName, its receiver is self new, and all the remaining cascaded messages thus go to the result of self new. This calls for a look at another aspect of cascading: Although all the remaining messages go to the same object, the state of this object changes as the message executes. At the beginning, the new object is presumably uninitialized. After firstName: firstName, its firstName component is initialized to firstName and the following lastName: message thus goes to this modified object. After lastName: the value of lastName is changed and the id: message thus executes on the receiver in this again different state.

This may raise the question why we call all these receivers 'the same object'. The answer is that the receiver is the same in the same sense that I am the same person as I was yesterday (but one day older and surely different in some other ways as well), that a moving car is the same object as the same car when it was parked, and so on. The concept of object identity is important and we will return to it later.

Main lessons learned:

- Transcript is a part of Visual Launcher that can be used to output text. It is often used to output intermediate results during testing.
- The most important messages to the Transcript object are clear, cr, show:, and tab. The show: message requires a string argument.
- Any object can be converted to a string, by the printString message.
- Cascading is a shortcut technique allowing you to leave out the name of the receiver when a sequence of consecutive statements uses the same receiver.
- To cascade, include the receiver in the first statement but leave it our in the following statements and replace the periods separating the consecutive statements with semicolons.
- Cascading is often misunderstood and this may be why it is not used more frequently. The precise rules for cascading are as follows:
  Cascaded expressions are separated by semicolons.
- All cascaded message have the same receiver as the last message *executed* before the first semicolon.
- Although all cascaded messages are sent to the same receiver, the state of the receiver may change as the cascade executes.

- Cascading without carefully checking who is the repeated receiver is a frequent cause of errors.

Exercises

1. Message printString works by passing storeOn: down the structure of composite objects. This illustrates the operation of messages and we suggest that you trace the execution of the following expressions and write a description of the execution of the printString message focussing on the passing of control to subordinate storeOn: receivers.
   a. self halt. (13@14) printString
   b. self halt. 'abc' printString
2. Predict and then confirm the result of 5 factorial factorial and of 5 factorial; factorial. Repeat for 5 + 5 * 5 and 5 + 5; * 5.
3. In the following code fragment, add code necessary to clear the Transcript and display the value of x on a new line after each statement. Use cascading whenever possible. Execute the program with *do it*.
   ```
   | x |
   x := 13.
   x := x + 3 * 5.
   x := x / 5.
   x := factorial
   ```
4. Repeat the previous exercise but type the code of the program into the Transcript view itself and execute it with *do it*. Then explain why it is better to use the Workspace to execute code fragments.
5. Write an expression solving the following problem and test it.
   a. Ask the user to select a rectangle on the screen and print the coordinates of its upper left and lower right corners in the Transcript, one point per line with a proper legend. (Hint: Look at class Rectangle.)
   b. Ask the user to enter the price of an item, calculate its provincial and federal tax according to Exercise 3.c of Section 3.5, calculate the final price, and print all input information and all results in the transcript.
6. Underline the receivers and give the results of the following cascaded messages. Explain.
   a. 3 + 4 factorial; squared; sqrt
   b. 3 factorial + 4 factorial; squared; sqrt
   c. 3 between: 4 and: 5; + 7
   d. 3 factorial between: 4 and: 15; + 7
   e. 3 factorial between: 4 factorial and: 15 squared; + 7 squared
   f. (Rectangle origin: 10@20 corner: 50@50) extent: 100@150; height: 40; width: 70; top: 40
7. This example shows that although the identity of the receiver does not change as it is reused by consecutive cascaded messages, its state may change. Use the Debugger's inspector to find the consecutive states of the receiver after each message:
   'abc' at:1 put: $x; at: 2 put: $y; at:3 put: $z; printString

3.10 Global variables, class instance variables, and pool dictionaries

In this section, we will introduce the remaining kinds of variables available in Smalltalk, starting with the following question what kind of object is Transcript. From its spelling, you would probably conclude that Transcript is the name of a class because it starts with a capital letter. In reality, Transcript is not the name of a class but the name of a *global variable*.

A global variable is a variable that is accessible from any Smalltalk code - any code fragment or any method in any class. This is quite unlike *temporary variables* which can be accessed only in the method in which they are declared, or *instance variables* which can be accessed only by instance methods of the class in which they are declared or its subclasses, or *class variables* which can be accessed only by instance and class methods in the class where the variable is declared or inherited. In other words, global variables have *global scope* whereas the scope of temporary, instance, and class variables is limited.

You are now probably wondering how Smalltalk knows about global variables if they are not defined in classes. The answer is that Smalltalk keeps the names of all globally accessible identifiers in a

special 'dictionary' object called Smalltalk[7]. This dictionary contains a list of 'keys', and for each of them its 'value' object. The keys are the globally shared identifiers, and the 'values' are their values. To see the contents of this dictionary, type and select the word Smalltalk and 'execute' it with *inspect*. You will get the inspector in Figure 3.15.

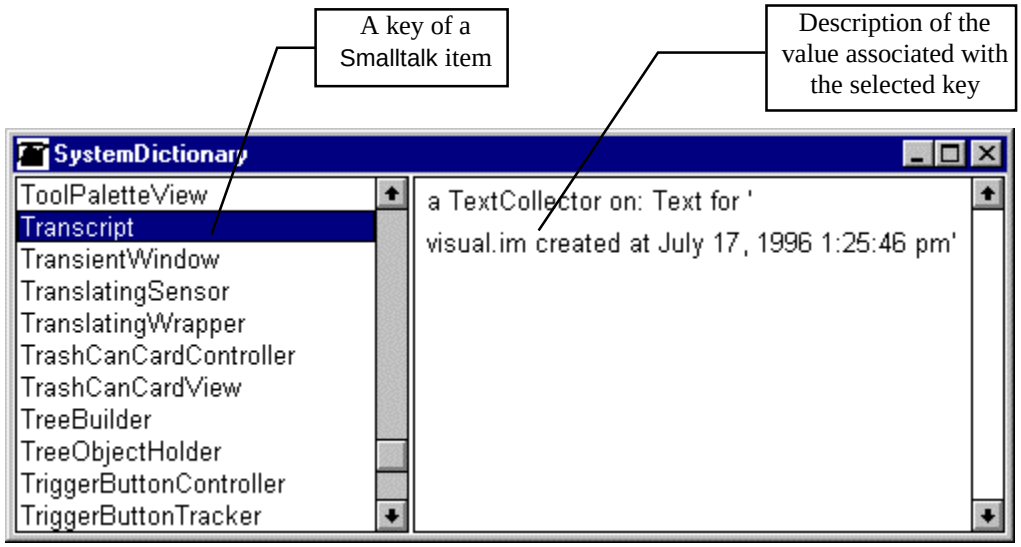

Figure 3.15. Inspector on the Smalltalk system dictionary obtained by executing *inspect* on Smalltalk.

The vast majority of keys in the Smalltalk dictionary are names of classes (in a sense, classes are also global variables because they are globally accessible) and their values are the class objects themselves. A few keys, such as Transcript, however, are global variables and their values are other kinds of objects. As an example, the value of key Transcript is an instance of class TextCollector, a class containing the definitions of clear, show: and all the other messages understood by Transcript. It is interesting to note that the identifier Smalltalk which refers to the Smalltalk dictionary is itself a global variable stored as a key in the Smalltalk dictionary.

How can you recognize a global variable? This is a tricky question but you could say that a global variable is a shared variable whose value is not a class. How can one create a global variable? One way is to type its name - for example NewGlobal - into some text space such as a Workspace, select it, and execute *do it* or *inspect*. This will open the notifier in Figure 3.16. If you select *global*, the identifier will be entered into the Smalltalk dictionary with nil as its value. You can then assign its value any time.

Figure 3.16. Creating a global variable via a notifier.

[7] We will talk about dictionaries in Chapter 10.

You can also create a global variable with an initial value by evaluating an assignment such as

```
NewGlobal := 10
```

which will open a Notifier as in Figure 3.16 and create and initialize the variable upon confirmation.

Another way to create a global variable is to add it via the <operate> menu of the inspector of the Smalltalk dictionary; You can then enter its value in the right half of the inspector window and *accept*. To remove a global variable, use *remove* in the <operate> the menu of the Smalltalk inspector.

When are global variables used? The answer is - very rarely. The advantage of global variables is that they persist from one saved Smalltalk session to another and are restored when you open Smalltalk again. (This explains why you get your Transcript and your classes with their class variables back when you open a new session.) Because of their persistence, global variables can be used to hold objects that should last beyond the execution of a single application or code fragment. The Transcript object is a perfect example because it maintains a text view with a record of system activities that you may want to read when you reopen Smalltalk.

However, global variables also have disadvantages: For one thing, since persistent objects are carried along throughout the whole session and for all remaining sessions until you remove them, they may occupy a lot of memory. And if you forget about them, this memory is wasted. Another disadvantage of global variables is that they violate one of the basic principles of Smalltalk - encapsulation. According to the principle of encapsulation, the scope of accessibility of objects should be minimal. Global variables are accessible globally and control over their values is available all over the whole system. This makes them vulnerable and unsafe because their values can be changed uncontrollably by any method or code fragment. These two disadvantages of global variables are a sufficient reason why their use is strongly discouraged and why the Smalltalk system itself defines only a few of them.

In addition to its fundamental role of holding all existing classes, the Smalltalk object is very important because it is the root from which all checks for live objects begin: All objects directly or indirectly referred to by Smalltalk are live, all other objects can be garbage collected. More formally, we can define *active objects* as follows:

1. All objects accessed from the Smalltalk dictionary are active.
2. All objects referenced by active objects are active.

As a consequence of this definition, all class objects are active (and will not be ‘garbage collected’). All objects referenced by classes (such as class comments, class variables, and class definitions) are also active because they are referenced by classes. All pool dictionary objects are also active. And all objects in pool dictionaries are active because they are referenced by pool dictionaries. All objects referenced by global variables are active because global variables are in Smalltalk. (This includes, among other things, all windows on the screen.) And all objects referenced by global objects are active, and all objects referenced by them are also active, and so on. If you can ‘find a path’ from the Smalltalk dictionary to an object, that object is active - and all other objects are inactive and will be garbage collected when the garbage collector is activated. Thousands of objects are normally active at any time.

By the way, the definition of an active object is said to be *recursive* because its second part defines what the concept *active object* means in terms of the concept *active object* itself. You could say that a recursive definition ‘feeds on itself’. Recursive definitions are very important and we will encounter them again later.

Pool dictionaries

In addition to global variables and classes, the Smalltalk dictionary contains a few specimens of another kind of shared object - a *pool dictionary*. A pool dictionary is a dictionary (with keys and values - like Smalltalk) whose *keys* are directly accessible to a selected group of classes - those classes in which it is defined as a pool dictionary. Any method in each class in this group can directly access each pool dictionary key as if it were a class variable, just like a class or instance variable. Other classes can also access pool variables (because pool names are keys in Smalltalk) but not directly – their access requires going through

the pool dictionary's name. To create a pool dictionary, add its name to the Smalltalk dictionary as for a global variable, enter values into it (we will see how in the chapter on dictionaries), and enter the name of the pool dictionary as the pool dictionary keyword in each class that should have access to it.

As an example of a pool dictionary, TextConstants contains information needed for display of text. One of its keys is, for example, called Tab and its value is the code of the character that produces a tab in text. All classes that define TextConstants as their pool dictionary can access its Tab directly. As an example, TextConstants is a pool dictionary in class Text (Figure 3.17) and you can thus execute messages such as

```
char = Tab          "Check whether variable char contains the Tab character."
```

in any method defined in class Text or its subclasses. Defining TextConstants as a pool dictionary eliminates the need to duplicate the information in each class that needs it, which would otherwise be necessary because these classes are not on the same branch of the hierarchy tree and cannot define these values as shared class constants. TextConstants provides a typical use of pool dictionaries – as a depository of useful constants whose access is faster than accessing them via methods.

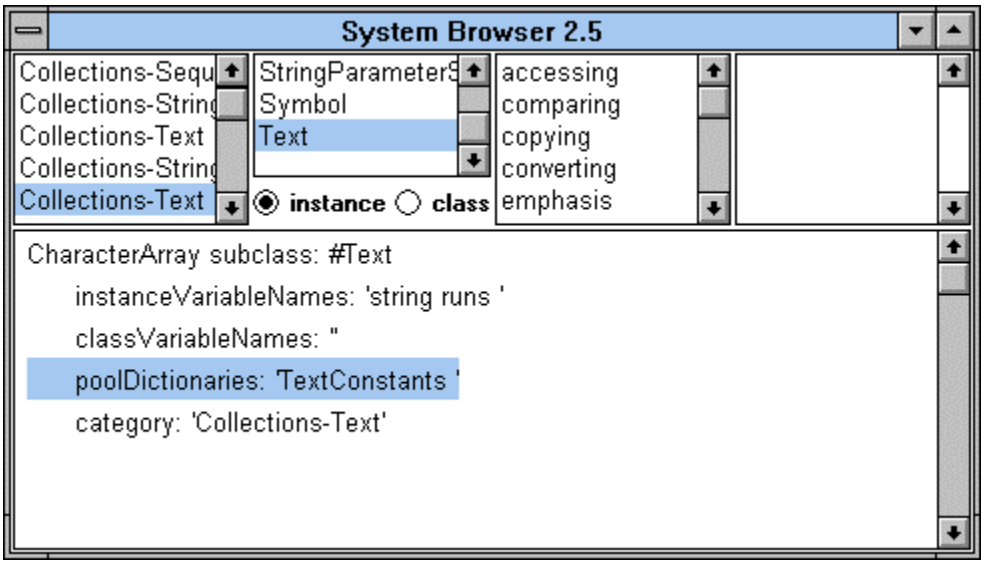

Figure 3.17. Class Text has a pool dictionary variable TextConstants.

To find where pool dictionary is used, open the browser on one of the classes that defines it, select the name of the pool dictionary in the definition, and execute *explain* in the <operate> menu. To find where a particular key (such as Tab) of a pool dictionary variable is used, open an inspector on the pool dictionary, find the pool dictionary variable key, and execute *references* from the <operate> menu.

Pool dictionaries have similar disadvantages as global variables, are rarely used, and their use is discouraged.

Class in

The dif
variable
private
the Brov

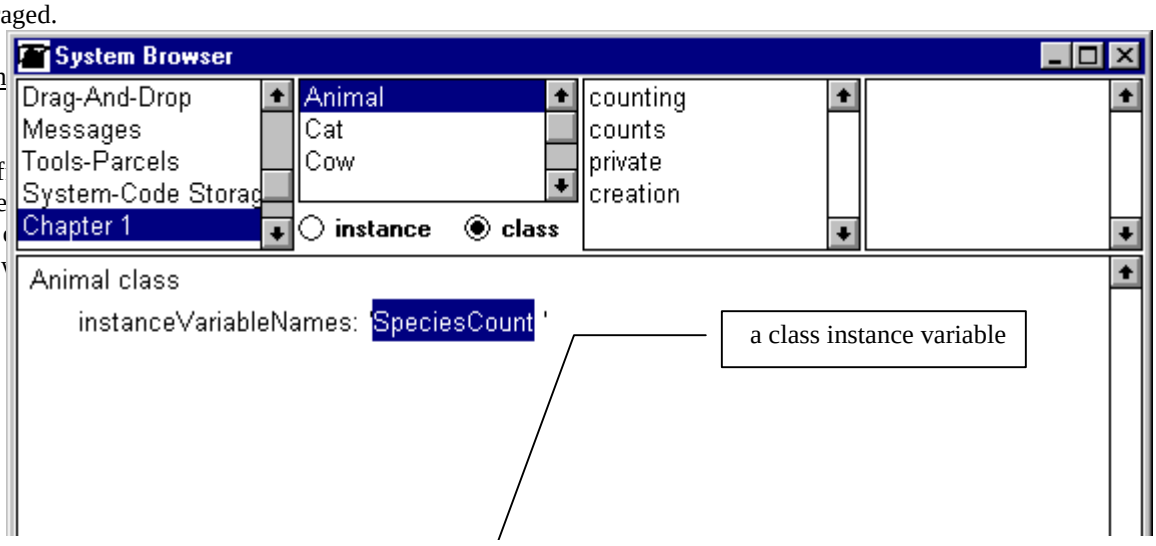

Figure 3.18. Class instance variables are listed on the class side of the browser.

Although class variables are rare, they are useful and *Farm 6* gives an example that illustrates their utility and the difference between class variables and class instance variables (Figure 3.19). The problem addressed by Farm 6 is counting how many cats, how many cows, and how many dogs a farm has, and how many animals there are on the farm altogether. To do these calculations, we defined two variables in class Animal. One is a class variable called TotalCount and you can find it on the instance side of the definition of Animal. The other is a class instance variable called SpeciesCount and you will find it on the class side of the definition of Animal in the Browser as shown in Figure 3.18.

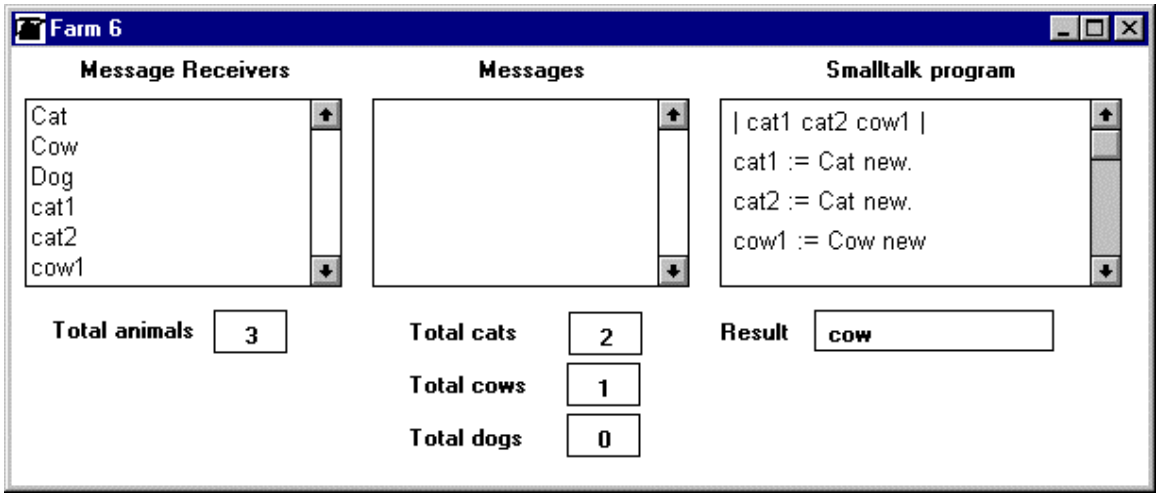

Figure 3.18. Farm 6 uses class variables and class instance variables for counting.

Both variables are inherited by Cat, Cow, and Dog and we increment them both when Animal gets the new message. Since all animals share the same copy of the class variable TotalCount, TotalCount is incremented every time that a new animal is created, whether it's a Cat, a Cow, or a Dog. As for class *instance* variable SpeciesCount, each animal class inherits its own *private* copy which means that Cat's SpeciesCount is incremented only when a new Cat is created, a Cow's SpeciesCount is incremented only when a new Cow is created, and a Dog's SpeciesCount is incremented only when a new Dog, is created. In this way, TotalCount keeps count all animals whereas SpeciesCount keeps count for each animal separately. This is a typical use for class variables and class instance variables.

Main lessons learned:

- A global variable is a variable accessible by any Smalltalk code and its description is stored in Smalltalk, the System Dictionary. Smalltalk is a global variable itself.
- Smalltalk elements are key-value pairs and all their keys are globally accessible across all Smalltalk code.
- Shared variables include names of classes, pool dictionaries, and global variables. Their identifiers start with a capital letter.
- The Smalltalk object is persistent, surviving beyond the execution of a program fragment or a Smalltalk session. Its contents are saved in the image file whenever you save the Smalltalk environment, and restored when you start a new session. All variables stored in it and their values are thus also persistent.
- Class instance variables are similar to class variables but their value is private to a class. This is different from class variables whose values are shared by the class and all its subclasses.

- A pool dictionary is a dictionary of variable-value pairs. Its variables can be directly accessed by all classes that define the dictionary as their pool dictionary. Its typical use is as a depository of useful constants shared by several classes unrelated in the class hierarchy.
- Although global variables and pool dictionaries have their advantages, their use beyond a few well-justified situations is discouraged because they violate encapsulation and require continuous allocation of significant memory space.
- An object is active if a link to it can be traced from the Smalltalk dictionary.
- Active objects are those that have a connection to Smalltalk. All other objects are inactive and will be garbage collected.
- A definition is recursive if it defines a concept in terms of itself.

Exercises

1. Inspect Smalltalk to find all references to Transcript.
2. Inspect the global variable Smalltalk in the Smalltalk inspector.
3. Inspect TextConstants with the inspector and list five interesting components.
4. Find all classes that define TextConstants as their pool dictionary variable. Find all references to key Tab in the TextConstants pool dictionary variable.
5. Examine 20 randomly selected keys in Smalltalk and find their values. How common are pool variables and global variables compared to classes?
6. Add global variable NewGlobal with value 7 to the Smalltalk and check that you can access it from any Workspace and from the Transcript. Change its value to 35. Save the image and exit Smalltalk, reopen Smalltalk and check that NewGlobal is still there with the same value, delete it, exit with saving, restart, and check that NewGlobal is gone.
7. Are any other kinds of shared objects stored in the Smalltalk dictionary?
8. Use *Farm 6* to find out about class and class instance variables.
9. Enact the *Farm 6* program shown above, assigning the role of Animal, Cat, Cow, and Dog classes to individual students and focusing on the behavior of TotalCount and SpeciesCount.
10. Examine 30 randomly selected classes and count how many of them have a class instance variable.

Conclusion

A Smalltalk code fragment consists of one or more statements separated by periods. Statements may be message send or assignment expressions.

Names of variables and methods are called identifiers. They are formed according to the following rules:

- An identifier must begin with a letter and the remaining characters must be letters or digits.
- In principle, identifiers may contain any number of characters. In reality, there is an implementation limit on the length but this limit is very high.
- Identifiers are case sensitive.
- Names of methods and temporary and instance variables start with a lowercase letter, names of classes and other shared variables start with an uppercase letter.
- The following words are reserved for special uses and cannot be used for identifiers: self, super, true, false, nil, and thisContext.

Smalltalk programmers choose identifiers very carefully to make programs as readable as possible. For additional documentation, use comments. Any text surrounded by double quotes is a Smalltalk comment and is ignored when the program is executed.

Smalltalk recognizes three types of messages - unary, binary, and keyword. Their names are formed according to the following rules:

- A unary message consists of an identifier and does not have any arguments.
- A binary message consists of one or two special symbols selected from the + - / \ * & @ , > < = and other special characters and has one argument. Symbol – combination – is not allowed.
- The selector of a keyword message consists of one or more keywords and each keyword is an identifier followed by a colon. In a message, each keyword must be followed by an argument.
- All unary messages and all keywords begin with a lowercase letter.

Selector refers is the full name of a message without arguments. Selectors are used in browser searches and in certain programming contexts.

To make it possible to refer to an object in several places in a code fragment or a method, assign it to a temporary variable using an assignment. Don't assign an object to a variable unless you need to refer to

it later in the program or unless the variable will make the code more readable. All variables must be declared before the first statement.

The scope of a temporary variable (the range in which it can be used) is the code fragment or method in which it is declared. An object bound to a temporary variable will be destroyed when the execution of the code in which the variable is defined is finished, unless it is bound to another object which survives.

In addition to temporaries, Smalltalk defines several other kinds of variables. Instance variables capture object state and each instance of a class has its own private copy; instance variables are inaccessible to other objects. Class variables are shared by the class in which they are declared, its instances, its subclasses, and their instances.

The lifetime of objects referenced by instance and class variables is the same as the lifetime of their parent objects. Since classes are persistent objects (survive from one saved Smalltalk session to another), objects bound to class variables are also persistent.

Code fragments are used to test pieces of code before they are used in methods. Although code fragments can be executed in any text view, Smalltalk programmers normally use a Workspace window for this purpose. To execute a code fragment, select it using the <select> button, and execute it using *print it*, *inspect*, or *do it* via the <operate> button. For execution, Smalltalk converts a code fragment to the equivalent of a temporary method and in this sense, a code fragment is thus a method.

Understanding the rules of formation and evaluation of Smalltalk expressions is essential for reading and writing Smalltalk code. The rules are simple: Everything is evaluated from left to right, parenthesized expressions first, unary messages next, binary messages next, and keyword messages last.

When a series of consecutive expressions are sent to the same receiver, Smalltalk programmers use cascading. Cascaded expressions are separated by semicolons and the receiver is not repeated. All expressions are executed by the receiver of the last message executed before the first semicolon.

Global variables are globally accessible, which means that they can be accessed from any method or code fragment. Pool dictionaries can are global in the same sense but their components can be accessed directly from classes in which they are named as pool variables. Class instance variables are similar to class variables but they are not shared by the defining class and its subclasses; instead, each of them has its own copy. Global variables, pool variables, class variables, and class instance variables are classified as shared variables which means that they can be accessed by more than a single class or its instance. Names of all shared variables begin with a capital letter and their values are stored in the Smalltalk System Dictionary. They are all persistent because they are kept in Smalltalk during the whole session, saved by *save as* and *save and exit*, and restored when a new session is opened.

All objects that can be traced from the Smalltalk dictionary are active. All other objects are inactive and will be collected by the automatic garbage collector.

Terms introduced in this chapter

assignment - consists of variable name, assignment symbol, and an expression; assigns binding to result of expression to the variable; returns the value of expression

binary message - message whose selector consists of one or two special symbols; has one argument

binding - link between a variable and the object attached to it

blue button - old name for <window> button

call stack –list of currently active messages in the order in which they have been activated

cascaded message - a statement consisting of a receiver and expressions separated by semicolons; all expressions are sent to the receiver of the last message executed before the first semicolon

case sensitivity – distinguishing the difference between upper and lower case characters in identifiers; Smalltalk identifiers are case sensitive

class instance variable - similar to class variable but each subclass has its own private copy

class variable - variable associated with a class and accessible to all its instance, class, and subclass methods; a single copy of the corresponding object is shared by all objects that can access it

code fragment - one or more statements constructed for testing purposes; an intermediate step in developing Smalltalk applications; in execution, treated as a method

compilation - the process of converting Smalltalk code into internal computer representation for execution; automatically performed when a code fragment is executed or method definition *accepted*

comment - explanatory note inserted into a program for documentation, has no effect on program execution; in Smalltalk, any text surrounded by double quotes

Debugger - tool used to analyze malfunctioning programs or to gain understanding of the behavior of code execution; provides step-by-step evaluation and access to variables

decompilation – the reverse of compilation

do it - <operate> menu command; executes the selected (highlighted) code

global variable - variable accessible from any method; stored with its value in the System Dictionary

identifier - name of variable, message, or keyword; Smalltalk identifiers are case sensitive, must begin with a letter, and contain only letters or digits; the length of a Smalltalk identifier is unlimited

inspect - <operate> menu command; executes selected code and opens Inspector on the result

Inspector - window for viewing and editing the components of an object

instance variable - internal variable of an instance of a class; a separate copy is owned by each instance and is not directly accessible to other objects

keyword – identifier followed by a semicolon

keyword message - message whose name consists of a sequence of keywords followed by arguments

nil - special object bound to variables before they are assigned a value by an assignment; instance of class UndefinedObject

object lifetime - span of time during which an object is active and accessible to other objects

persistence - object's ability to survive from one session to another; Smalltalk objects are not persistent unless they are stored in Smalltalk or directly traceable to it

pool dictionary – a globally accessible dictionary of variable names and values; variables are accessible directly in classes that specify the dictionary as their pool dictionary

print it - <operate> menu command; executes the selected code and prints a description of the result

recursive definition - definition defining a concept in terms of itself

red button - old name for <select> button

rules of evaluation - rules used to determine the order of execution of components of a combined message

scope - range of code in which an identifier is recognized

selector - the name of a message as used in message searches; does not include arguments

shared variable - variable accessible to more than a single object; includes global variables, class variables, class instance variables, pool dictionary variables, and pool dictionary variables

Smalltalk - the single instance of SystemDictionary holding the names and values of all shared variables; saved with every save operation and restored when a Smalltalk session is opened

statement - expression followed by a period, or the last expression in method definition

temporary variable - variable defined at the beginning of a method

Transcript - text view attached to the bottom of the Visual Launcher; used by the system to output certain messages and by programmers to output intermediate results during testing

unary message - message consisting of an identifier; does not have any arguments

variable - identifier used to refer to an object; initially automatically bound to *nil*

variable declaration - list of all variables used in a method; must precede the first statement

<window> button - the rightmost button of the mouse

<window> menu - pop up menu of the <window> button; contains window-related commands

Workspace - window used by Smalltalk programmers to execute code fragments

yellow button - old name for <operate> button