Chapter 4 - True and False objects, blocks, selection and iteration

Overview

In this chapter, we begin exploring the most important classes of Smalltalk. We start with classes representing true and false objects because they are needed to make choices and to control repetition, and this is required in almost all programs.

The use of true and false objects heavily depends on the concept of a block. In essence, a block is a sequence of one or more statements that are always executed together. In the case of program choices and iteration, each alternative action or sequence of iterated statements is represented by a block which is executed or skipped depending on the result of evaluating a condition with a true or false result.

4.1. Why we need true and false objects

Most problems include questions that must be answered to determine what to do next. As an example, the response to a request for a book from a library catalog depends on the answer to questions such as 'Is the book in the library?' and 'Is the borrower allowed to take the book out?'. As another example, a program printing paychecks processes a list of employees, constantly asking the question 'Is there another employee record to print?' When the answer is 'no', processing stops.

In English, answers to questions such as these are usually in the form of 'yes' and 'no' but programming languages use the terms 'true' and false'. The two forms are equivalent because a question such as 'Is the book in the library?' can be formulated as 'Is it true that the book in the library?' and 'yes' and 'no' answers then naturally translate to 'true' and 'false'.

Smalltalk 'true' and 'false' objects are implemented by classes True and False. These two classes share many properties and they are thus defined as subclasses of the abstract class Boolean (Figure 4.1). The name Boolean or 'logic' is also used when we talk about the true and false objects in general[1].

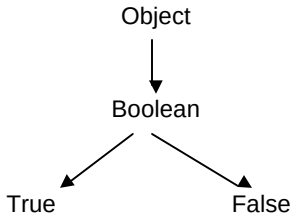

Figure 4.1. Booleans in Smalltalk class hierarchy.

As an illustration of the use of Booleans, consider a program that calculates tax. Assume that earners of incomes up to $7,000 pay no income tax while higher income earners pay 7% of their income above $7,000. The tax calculation program will read the income and ask: 'Is it true that the income is not above $7,000?' If the answer is 'true' the program will take one course of action (tax = 0) but if the answer is 'false' the program will calculate a tax. This could be implemented as follows:

1. Set tax to 0.
2. Is income is greater than $7,000? If the answer is 'true', change tax to 7% of income over $7,000.

In a more realistic tax program there would probably be more tax brackets such as no tax up to $7,000, 7% on amounts between $7,000 and $15,000, 12% on amounts between $15,000 and $30,000, and 25% on amounts exceeding $30,000. The logic of this calculation is more complex but still based on true/false questions:

[1]The name Boolean is used to honor the 19th century British mathematician George Boole who developed mathematical foundations of logic.

1. Initialize tax to 0.
2. Is income between 0 and 7,000? If the answer is 'true', calculation is finished.
3. Calculate 7% of income between 7,000 and 15,000 and add this amount to tax.
4. Is income less or equal to 15,000? If the answer is 'true', calculation is finished.
5. Calculate 12% of income between 15,000 and 30,000 and add this amount to tax.
6. Is income less or equal to 30,000? If the answer is 'true', calculation is finished.
7. Calculate 25% of income exceeding 30,000 and add this amount to tax.

As another example of the use of Booleans, a program that copies files from one disk to another might work on the following principle:

1. Is there is another file to be copied?
2. If the answer is 'false', execution is finished.
3. If the answer is 'true', copy the file and return to Step 1 to repeat the sequence.

These examples show that Boolean objects are needed to

- decide whether to take an action or not,
- decide which of several possible alternative courses of action to take, and
- decide whether to repeat a sequence of actions.

Class Boolean and its subclasses True and False are essential in all these operations and implement some of the corresponding methods. The remaining ones are left to a few other classes that will be introduced later.

Input of Boolean values

Before we introduce the messages that operate on Boolean objects, we will show how a user can input a Boolean value by using a *confirmer window*. To produce this window, class Dialog contains the class message confirm: which returns an instance of True (the true object) or an instance of False (the false object) depending on the user's choice. As an example,

```
Dialog confirm: 'Do you want to close the file?'
```

opens the window in Figure 4.2 and returns true if the user clicks *yes* or hits the <*Enter*> key; it returns false if the user clicks *no*.

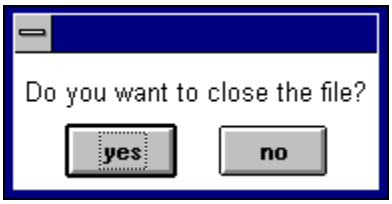

Figure 4.2. The confirm: message opens a confirmer window with a message-specified prompt.

Main lessons learned:

- Classes Boolean, True, and False are the basis for deciding whether an action should be taken or not, which of several actions should be executed, and whether an action should be repeated or not.
- Class Dialog provides message confirm: for input of Boolean values. It opens a confirmer window which solicits a true/false response in the form of a *yes* or *no* answer.

Exercises

1. Formulate an algorithm for finding the smallest of three numbers.
2. One way to calculate an integer approximation k of the square root of a positive integer n is as follows: Start with k = 1, and keep incrementing k by 1 until its square exceeds n. Subtract 1 from k to get the approximation. Formulate this principle as an algorithm.
3. Describe a situation where Booleans are needed to decide
   a. whether to execute an action
   b. which of two alternative actions to take
   c. whether to repeat a sequence of actions one more time
4. Write an expression to open a confirmer with the text 'Do you understand Booleans?' and execute it with *inspect* and *print it* to see the two possible results.
5. In addition to confirm:, Dialog contains a related message called confirm:initialAnswer:. Look it up in the System Browser and write an example expression to show how it works.

4.2 Boolean messages for deciding whether to take an action or not

The simplest use of Booleans is for deciding whether to take a specified action or not. Smalltalk contains two messages called ifTrue: and ifFalse: for dealing with this situation and calls them *controlling* messages. As an example,

```
(Dialog   confirm: 'Do you want to see 500 factorial?')          "Returns true or false."
          ifTrue: [Transcript clear; show: (500 factorial printString)]   "Executed for true receiver."
```

asks the user for a true or false answer and executes the bracketed statements if the result is true. The bracketed statements are ignored if the confirmer returns false. Message ifFalse: has the opposite effect. More formally, the two messages have the form

```
aBoolean ifTrue: aBlock
aBoolean ifFalse: aBlock
```

where aBlock is an instance of class BlockClosure, a sequence of statements surrounded by square brackets such as

```
[Transcript show: 'Test'; cr]
```

Evaluating a block returns executes all statements in the block and returns the result of the last statement executed in the block. (A more accurate description will be given later.)

Execution of ifTrue: can be described diagramatically by the *flowchart* in Figure 4.3 and ifFalse: follows the obvious alternative pattern.

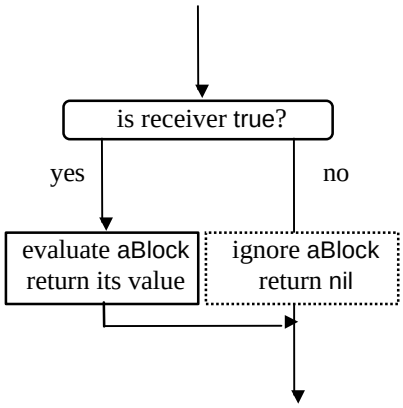

Figure 4.3. Evaluation of ifTrue: aBlock.

We will now illustrate ifTrue: and ifFalse: on several examples.

Example 1: Giving the user a choice

Problem: Write a code fragment to ask the user whether to delete a file or not. If the answer is 'yes', the open the *notifier window* in Figure 4.4.

Figure 4.4. Notifier window of Example 1.

Solution: To display a notifier window, send warn: to class Dialog as in

```
Dialog warn: 'You will not be able to recover the file later!'
```

The complete program is as follows:

```
| answer |
"Ask the yes/no question and store the answer in variable answer."
answer := Dialog confirm: 'Do you want to delete the file?'.
"If the answer is yes, display notifier window."
answer ifTrue: [Dialog warn: 'You will not be able to recover the file later!']
```

Type the program into a Workspace and execute it. Note that you don't have to type the ifTrue: keyword, just press <*Ctrl*> *t* (for 'true') and Smalltalk will insert the ifTrue: text; similarly <*Ctrl*> *f* produces ifFalse:. Note also that we could rewrite the code without the variable as follows:

```
"Ask the yes/no question and if  the answer is yes, display notifier window."
(Dialog confirm: 'Do you want to delete the file?')
        ifTrue: [Dialog warn: 'You will not be able to recover the file later!']
```

This form is quite readable and should not cause any misunderstandings; we thus prefer it.

Example 2: Converting text to uppercase letters

Problem: Ask the user to enter some text (a string) and convert it to upper case if the user so desires. Print the resulting text in the Transcript window.

Solution: A message to display a window and ask the user for a string is available in class Dialog and its most common form is request:initialAnswer:. As an example

```
Dialog request: 'What is your favorite beverage?' initialAnswer: 'Skim milk'
```

produces the window in Figure 4.5. If the user clicks *OK* or presses <Enter>, it returns the displayed initial answer *'Skim milk'*. If the user types another answer and clicks *OK* or presses <Enter>, it returns the new string. If the user clicks *Cancel*, it returns an empty string with no characters.

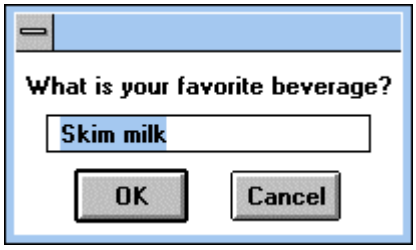

Figure 4.5. Dialog request: 'What is your favorite beverage?' initialAnswer: 'Skim milk'

To write the program, we need to know how to convert a string to upper case. if we don't, we open class String (that seems the logical place) and search for a suitable method by executing command *find method* in the <operate> menu in its instance protocol sview. Unfortunately, the list that opens does not offer any suitable method. We thus try String superclass CharacterArray and find that its method list (Figure 4.6) includes method asUppercase. When we open its definition , we find that it indeed solves our problem. (Messages converting one type of object to a related form of object are among the most frequently used Smalltalk messages. They usually begin with as, as in asUppercase or asNumber.)

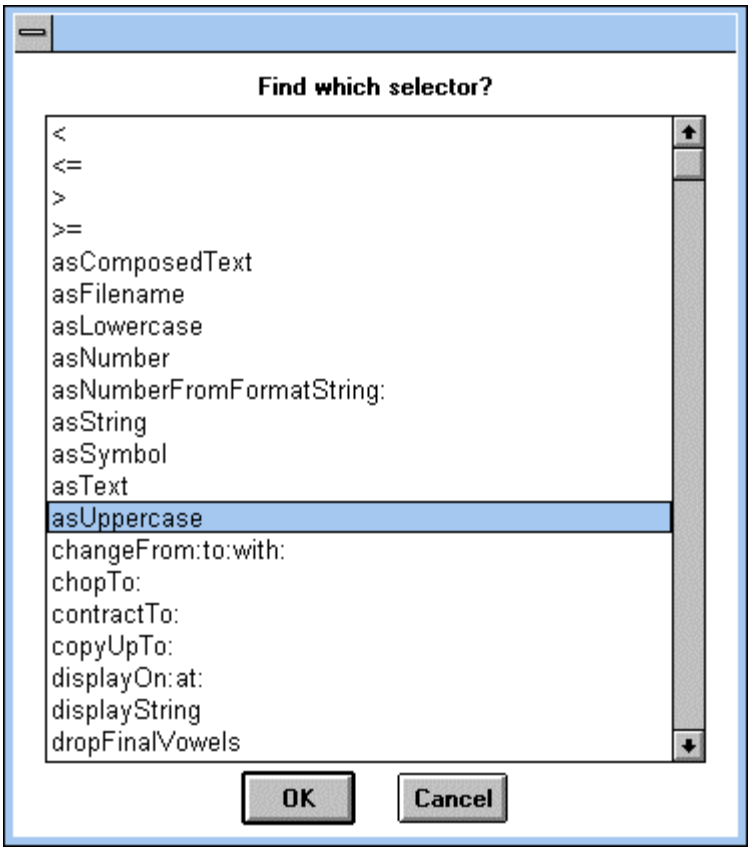

Figure 4.6. Response to *find method* in the instance protocol view of class CharacterArray..

We now have all necessary pieces to solve our problem and the complete solution is as follows:

```
| string convert |
“Get a string from the user.”
string := Dialog request: ‘Enter a string’ initialAnswer: ‘’.          “Display prompt without any initial string."
“Ask whether user whether to convert the string to upper case.”
convert := Dialog confirm: ‘Do you wish to convert the text to upper case?’.
“If the answer is yes, change string to upper case.”
convert ifTrue: [string := string asUppercase].
Transcript clear; show: string
```

Type the program into a workspace and execute it trying both ’yes’ and ’no’ answers to test how it works.

Main lessons learned:

- To control whether to execute a block of statements or not, use ifTrue: or ifFalse: from classes True and False. Both use a block as an argument.
- A block is a sequence of zero or more statements surrounded by square brackets. A block is an object, an instance of class BlockClosure. Evaluating a block evaluates the statements in the block and returns the result of the last statement.
- When the receiver of ifTrue: is a true object, the block argument is evaluated and the result returned. When the receiver is false, the block is ignored and nil is returned. Message ifFalse: has the opposite behavior.
- Messages converting one type of object to another are among the most frequently used Smalltalk messages. They usually begin with as, as in asUppercase or asNumber.

Exercises

1. Write a code fragment asking the user to choose whether to end program execution or not. If the answer is 'yes', nothing happens; if the answer is 'no', display a notifier saying 'Closing application.'.
2. Every statement using ifFalse: can be converted to ifTrue: and vice versa by inverting the condition that calculates the receiver. Rewrite the following statements with the alternative message:
   a. x > 3 ifTrue: [Transcript clear] "Rewrite using ifFalse:"
   b. x ~= y ifFalse: [Dialog warn: 'Wrong value of x'] "~= means 'not equal', 'equal' is ="
3. Check that when you replace the line
   convert ifTrue: [text := text asUppercase]
   in Example 2 with
   convert ifTrue: [text asUppercase]
   the program will not work. The reason is that expression text asUppercase returns a copy of the text object converted to uppercase *but does not change* the value of the receiver text itself. This is a common property of conversion messages that is frequently forgotten.
4. Count all conversion messages starting with as in the library that have the pattern as... message. Use *implementors of...* in the *Browse* command of the launcher with pattern as*. The * symbol is called a *wildcard character* and the search will use it to match any string starting with as such as asUppercase.
5. Some conversion messages don't have the form as* and we have already encountered one of them. Which one is it? (Hint: The message does not convert the receiver into a related object but produces an object that describes the receiver in a readable form.)

4.3 The definition of ifTrue:

The best way to understand how a message works is to study its definition. Let's examine method ifTrue: to see how it works. In class Boolean, the definition of method ifTrue: is as follows:

```
ifTrue: alternativeBlock
"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. This method is typically not invoked because ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."

    ^self subclassResponsibility
```

Since this is our first method, let us first explain its structure. A method definition starts with a *heading* which defines the selector and names the arguments, if any. In this case, the heading is

```
ifTrue: alternativeBlock
```

the selector is ifTrue: and the argument is called alternativeBlock. The name alternativeBlock indicates that the argument should be a block, and that its execution is an alternative, a possibility.

Following the heading is usually a *comment* describing the purpose of the method and possibly how it works. We will have more to say about the comment in this definition in a moment. The comment may be followed by the definition of temporary variables for the method (none in this case), and by the *body* of the method - the part specifying how the method works. The body consists of a sequence of zero or more statements. A more formal definition of the rules of Smalltalk (its syntax) is given in Appendix 8.

Every Smalltalk method return an object and its default value (the value returned returned unless the program specifies otherwise) is the receiver. As an example, if the receiver of the message is a true object, the method returns true by default. If we want to return something else, we must calculate the object and put the caret symbol (the return operator ^) in front of the expression that calculates the result - as in the definition of ifTrue:. The return operator also forces the execution of the method to stop at this point. This makes it possible to exit from any block anywhere inside a method, not just at the end.

After these general notes, let us now examine the body of the definition . The line

```
^self subclassResponsibility
```

says that the receiver of the message sends message subclassResponsibility to itself (object self) and returns the object returned by that method.

Message subclassResponsibility is defined in class Object and it is thus understood by all objects. It opens the Exception Window in Figure 4.7 telling you, in essence, that you cannot send this message to this object and that this method should be redefined in a lower level class. Message subclassResponsibility is frequently used in abstract classes (such as Boolean) when they leave the implementation of a shared message to concrete subclasses.

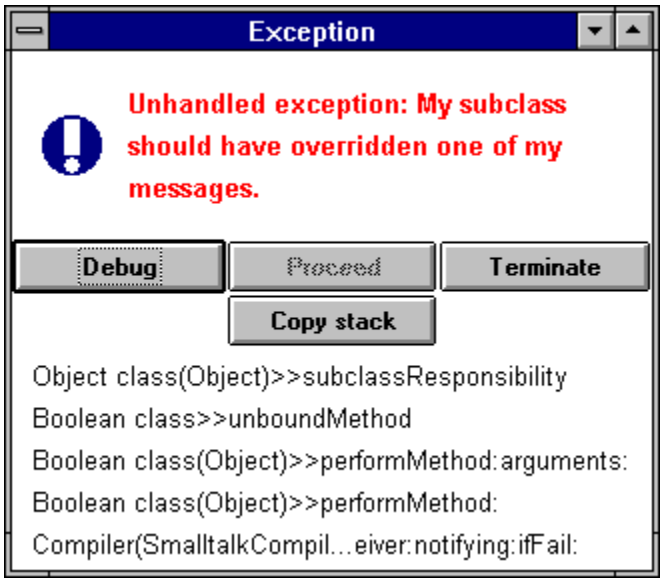

Figure 4.7. Result of trying to execute a message whose definition is left to a subclass.

The working definitions of ifTrue: are thus found in classes False and True. This is natural because class Boolean is abstract, no instances of it ever exist, and its definition of ifTrue: should thus never be executed.

We will now move on to subclasses, starting with False. In class False, message ifTrue: should *not* execute the alternative block and the definition is as follows:

```
ifTrue: alternativeBlock
"Since the condition is false, answer the value of the false alternative, which is nil. This method is typically
not invoked because ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."
        ^nil
```

In other words, the false object ignores the argument and returns the object nil. In class True, message ifTrue: aBlock must evaluate aBlock and the definition is thus

ifTrue: alternativeBlock
"Answer the value of alternativeBlock. This method is typically not invoked because ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."
^alternativeBlock value

This definition sends message value to alternativeBlock which forces the block to execute the statements inside the square brackets. The caret then returns the value calculated by the last executed statement.

Message value is very important and it is thus useful to give an example to explain how it works. As an illustration of the operation of the value message

[Transcript clear; show: 'Testing block closure'; cr. Dialog warn: 'A test'] *value*

evaluates the statements inside the block and returns the result. It has exactly the same effect as

Transcript clear; show: 'Testing block closure'; cr. Dialog warn: 'A test'

If this is so, why should we ever want to put statements in a block and send them value? The reason is that we sometimes cannot predict which statements we will want to evaluate – and message ifTrue: is an example of this: We know that if the receiver is true, the method will want to evaluate a block of statements but these statements can be anything so the best we can do is put them inside a block and evaluate the block with value.

In-line messages

The comment of ifTrue: contains a note about *in-line compilation*:

"This method is typically not invoked because ifTrue:/ifFalse: expressions are *compiled in-line* for literal blocks."

What does this mean? Messages ifTrue: and ifFalse: are among the most frequently used messages and because they are used so often, they must be implemented very efficiently. To achieve this, when the compiler encounters ifTrue: or ifFalse:, it does not create code to send the messages in the definition as it normally does, but inserts code to execute the sequence of statements in the block directly. This technique is used by a very small number of Smalltalk methods and is called in-lining. In the Debugger, in-lined messages are marked as *optimized*.

You might be wondering why Smalltalk bothers with the definition of ifTrue: if it is not really used. There are two reasons for this. The less important one is that the definition shows the exact effect of the message. The more important reason is that there are special situations in which the compiler does *not* create in-line code (because it cannot) and executes the message using the definition instead. We will explain how these situations arise later.

Objects true and false are unique

True and False are among the very few classes that allow only one instance; class UndefinedObject is another one, and small integers, symbols, and characters also have this property. Such unique instances are sometimes called singletons. The single instance of True is called true, the single instance of False is called false, and the single instance of UndefinedObject is called nil. When the compiler reads Smalltalk code, it recognizes these special words called *literals* and treats them differently. This means that when you need a true object (or a false or a nil), you can specify it by using the literal form true (or false or nil) directly in the code. In other words, you don't have to create, for example, instances of True by sending

True new

or a similar message. In fact, this statement would not work and would open an Exception Window telling you that you cannot create new True objects. As we mentioned, there are a few other kinds of literals such

as literal strings such as 'This is a your tax', literal numbers such as 132 or 3.14, and literal blocks such as [Transcript cr].

Main lessons learned:

- A singleton is an instance of a class that allows only a single instance to exist. Examples of singletons are objects true, false, and nil.
- A literal is a textual representation directly converted into an object by the compiler. Examples of literals include numbers, strings, nil, true, false, and block literals.
- When the compiler encounters a message that is compiled in-line, it does not execute the messages in the body of the method but creates machine code instead. Smalltalk uses in-lining to increase operation speed in a few very frequently used messages.

Exercises

1. Find all definitions of ifFalse: using the *Browse implementors of* command and explain how they work.
2. Explain the definition of the new message in classes Boolean, True, and False.

4.4 Selecting one of two alternative actions

In the previous section, we dealt with situations where the program must decide whether to execute a block of statements or not. In this section, we will deal with situations in which the program must select one of two alternative actions.

Example 1: Definition of max:

Class Magnitude defines the properties shared by all objects that can be compared. If two objects can be compared, we can find which one is larger and which one is smaller and Magnitude thus contains the definitions of methods max: and min: which can be used as follows:

```
3 max: 4        "Returns 4."
3 min: 4        "Returns 3."
```

The definition of max: is as follows:

```
max: aMagnitude
"Answer the receiver or the argument, whichever has the greater magnitude."
        self > aMagnitude
                ifTrue: [^self]
                ifFalse: [^aMagnitude]
```

Example 2. Decide whether three numbers form a right-angle triangle

Problem: Write a code fragment to prompt the user to enter three numbers, test the numbers, and print a message in the Transcript to say whether the numbers form the sides of a right-angled triangle or not.
Solution: If a, b, and c are the sides of a right angle triangle and c is the hypotenuse, then $c^2 = a^2 + b^2$. The algorithm for finding whether three numbers for a right-angle triangle is thus as follows:

1. Ask the user to enter the hypotenuse; call it c.
2. Ask the user to enter the next side; call it a.
3. Ask the user to enter the next side; call it b.
4. Clear the Transcript.
5. Check whether $c^2 = a^2 + b^2$.
6. If true, print 'triangle is right-angled'; if false, print 'triangle is NOT right-angled'.

To write the program, we need to know how to read a *number* from the keyboard and how to select one of two choices depending on the value of a Boolean receiver.

To read a number from the keyboard, read a *string* using request:initialAnswer:, and convert it into a number using the conversion message asNumber. To choose one of two alternatives use the Boolean message ifTrue:ifFalse:. Beginners often think that ifTrue:ifFalse: are two messages; in reality, it is a *single* message with two keywords, each expecting a block argument. Its operation is described in Figure 4.8.

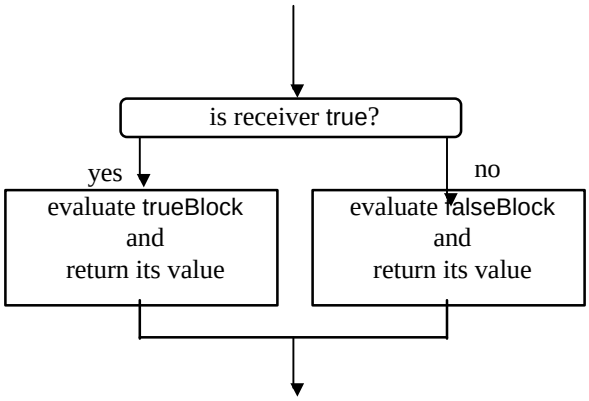

Figure 4.8. Evaluation of ifTrue: trueBlock ifFalse: falseBlock.

With this information, we can write the following first version of the program. We will see that this version of the program is not entirely correct but since the mistake is frequent and produces behavior that seems very confusing to a novice, we will start with the incorrect version.

```
|a b c |
c := Dialog request: 'Enter the length of the hypotenuse' initialAnswer: '' asNumber.
a := Dialog request: 'Enter the length of the second side' initialAnswer: '' asNumber.
b := Dialog request: 'Enter the length of the third side' initialAnswer: '' asNumber.
Transcript clear.
a squared + b squared = c squared
        ifTrue: [Transcript show: 'This IS a right-angled triangle']
        ifFalse: [Transcript show: 'This is NOT a right-angled triangle']
```

When we try to execute this program, Smalltalk opens the Exception Window in Figure 4.9. The problem looks very unpleasant because the displayed messages on the message stack don't look like anything in our code!

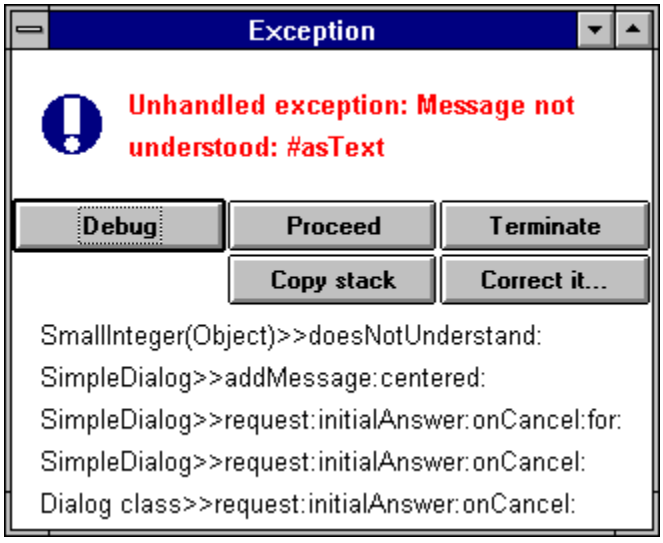

Figure 4.9. Exception Window produced by the first solution of Example 1.

We open the debugger to check where the problem occurred and find that our program is trying to execute message

```
request: 'Enter length of the first side - the hypotenuse' initialAnswer: '' asNumber
```

and fails.

Let's take a closer look at the evaluation of this line: The expression does not contain any parenthesized expressions. There is one unary messages - asNumber and the receiver is the empty string ''. The first thing that happens when the statement is executed is thus that Smalltalk takes an empty string and converts it to a number. This works and returns 0. (Method asNumber first initializes a temporary variable to zero and then keeps converting successive characters to numeric values until it reaches a character that is not a digit. It then stops and returns the result. In our case, there are no digits and the returned result is thus 0.) Our expression now effectively becomes

```
request: 'Enter length of the first side - the hypotenuse' initialAnswer: 0
```

This expression does not contain any more unary messages. There are no binary messages but there is a keyword message request:initialAnswer:. Its first argument is a string and the second argument is a number. Smalltalk starts executing this message and fails because the definition of request:initialAnswer: assumes that *both* arguments are strings. Now the message in the Exception Window makes more sense: It essentially says that an integer does not how to convert itself to a Text object.

To correct the problem, think about our goal: We first want to read a string using request:initialAnswer: and *then* convert it to a number. The correct formulation is thus

```
a := (Dialog request: 'Enter length of the first side - the hypotenuse' initialAnswer: '') asNumber.
```

Since our code fragment uses the same pattern throughout, we must modify the rest in the same way:

```
|a b c |
Dialog warn: 'You will be asked to enter three sides of a triangle. The first number is the hypotenuse'.
c := (Dialog request: 'Enter length of the first side - the hypotenuse' initialAnswer: '') asNumber.
a := (Dialog request: 'Enter length of the second side' initialAnswer: '') asNumber.
b := (Dialog request: 'Enter length of the third side' initialAnswer: '') asNumber.
Transcript clear.
a squared + b squared = c squared
        ifTrue: [Transcript show: 'This IS a right angled triangle']
        ifFalse: [Transcript show: 'This is NOT a right angled triangle']
```

This works but a note of caution is in order. When a condition is based on testing numbers, such as $c^2 = a^2 + b^2$, we will get the exact result when using integers (such as 3, 4, and 5) but we cannot be certain that the test will be evaluated accurately if the numbers are floating-point numbers such as 3.14 or -0.45. Computer arithmetic with floating-point numbers always performs conversion to binary and this conversion is almost always inaccurate. Numeric comparisons involving floating-point numbers should thus be avoided and if we must compare floating-point numbers, we should accept a tiny difference as equality. This problem is common to floating-point arithmetic in any language, not only in Smalltalk.

Example 3: Calculating the value of a function with a complicated definition

Problem: Obtain a number from the user and calculate and print the argument and the value of a function defined as follows: If the argument is negative, the result is 0. If the argument is greater or equal to 0 and less or equal to 10, the result is equal to the original argument. If the argument is greater than 10 and less than 100, the result is the square of the argument. For all other argument values, the result is the cube of the argument. Print the original argument and the result in the Transcript. Sketch the graph of the function.

Solution: The solution consists of a sequence of consecutive tests directly reflecting the definition:

```
| argument result |
argument:= (Dialog request: 'Enter a number' initialAnswer: '') asNumber.
argument < 0
        ifTrue: [result := 0]
        ifFalse: [argument <= 10                                        "Aargument is >= 0."
                        ifTrue: [result := argument]                    “Argument is between 0 and 10.”
                        ifFalse: [argument < 100                        "Argument must > 10."
                                ifTrue: [result := argument squared] “Argument between 10 and 100.”
                                ifFalse: [result := argument * argument * argument]]].  “Over 100.”
“Print the argument and the result.”
Transcript clear;
        show: 'Argument ', argument printString;
        tab;
        show: ' Result ',result printString
```

This example required nesting the tests one inside the other and in such a case, matching of brackets is sensitive. If you are not sure whether your brackets match, click twice just after an opening bracket or just before a closing one and the text between the brackets will be highlighted.

Example 4: The definition of ifTrue:ifFalse:

The definition of ifTrue:ifFalse: *in class True* must evaluate the first block argument and ignore the second and this is exactly what happens:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
"Answer the value of trueAlternativeBlock. This method is typically not invoked because ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."

        ^trueAlternativeBlock value
```

Can you use this model to write the definition of ifTrue:ifFalse:in class False without looking? The library also contains an ifFalse:ifTrue: method which has a similar effect and is included only as a very rarely used convenience.

Main lessons learned:

- To select one of two alternative blocks of statements, use ifTrue:ifFalse: or ifFalse:ifTrue:.
- Comparison based on floating-point numbers is almost always inaccurate and should be avoided.

Exercises

1. Write a code fragment to request a number, ask whether it should be negated, and print the result in the Transcript window. (Hint: Use unary message negated; putting a minus sign in front of a variable as in x := -y will not work - try it.)
2. Implement the multiple tax bracket problem from Section 4.1 and print the input, intermediate results, and total tax in the Transcript.
3. Display all references to ifTrue:ifFalse: and to ifFalse: ifTrue: and comment on your findings.
4. If you must compare two floating-point numbers, you must be prepared for a small inaccuracy. You can, for example, say that x and y are ‘equal’ if their difference is smaller than one millionth of the smaller of the two numbers. Modify Example 2 along these lines. (Hint: Use message min:.)

4.5 Use ifTrue: and ifFalse: only when necessary

Probably the most important rule of programming is that code should be as simple as possible and one of the most common violations of simplicity is the use of selection messages. In Smalltalk, fortunately,

you can usually use polymorphism instead of ifTrue:/ifFalse:. We have already mentioned this when we discussed polymorphism and we will now briefly restate the example.

Assume that we want to write a drawing program for drawing various geometric shapes. One possible solution is to define a Pen class with a drawing method draw:at: along the following lines:

```
draw: aGeometricShape at: aPoint
"Draw aGeometricShape at the specified point of the screen."
aGeometricShape isMemberOf: Line          "Check whether the receiver's class is Line."
        ifTrue: [aShape drawLine at: aPoint].
aGeometricShape isMemberOf: Rectangle
        ifTrue: [aShape darwRectangle at: aPoint].
aGeometricShape isMemberOf: Circle
        ifTrue: [aShape drawCircle at: aPoint].
aGeometricShape isMemberOf: FreeLine
        ifTrue: aShape drawFreeLine at: aPoint].
etc.
```

With a reasonable number of geometric shapes, this method will be long and awkward. It will also be inefficient because drawing a shape will require testing chain of ifTrue: messages, wasting a lot of time just finding which part of the code should be executed. Moreover, adding a new shape will require adding code to this method and any other method that involves some kind of drawing, and forgetting to modify one of them is a distinct possibility. A much better solution is to define a special drawing method drawAt: for each kind of geometric object. The Pen's drawing method is then simply

```
draw: aGeometricShape at: aPoint
"Draw aGeometricShape at the specified point of the screen."
        aGeometricShape drawAt: aPoint
```

This is much simpler, much easier to understand, and faster. Besides, if a new kind of geometric object is added, this method does not have to be changed at all. But is it always possible to eliminate tests? Obviously not. If we want to ask the user whether to delete a file or not, we must post a yes/no answer and decide using ifTrue:/ifFalse:. Similarly, if we want to calculate tax in a multiple tax bracket system, we must test whether the numerical values are in certain ranges or not. There are many situations like this but if we can avoid ifTrue:/ifFalse: by using polymorphism, we always should.

Main lessons learned:

- Using ifTrue:/ifFalse: indiscriminately leads to complex and inefficient code that is difficult to modify.
- Use polymorphism instead of ifTrue:/ifFalse: whenever possible.

4.6 Creating a new class and a new method

Up to this point, we have been experimenting with small code fragments but programming in Smalltalk consists of creating classes and methods. Now that we know what a method looks like, let's write one.

Example 1: Class Name and its creation, initialization, and accessing methods

Although this example does not have anything to do with Booleans, we think that it is useful because it shows how to create classes and methods on the simplest possible example. We will develop more relevant examples later.

Assume that we have an application that requires personal names and that we need to distinguish between first name, middle name, and last name. This means that we cannot group all parts of a name into a single string and to keep the parts separated, we must assign one instance variable to each. We will call these instance variables firstName, middleName, and lastName. With this, we are now ready to define the

class. To do this, open a Browser, and add a new category called for example Tests using the *add …* command in the <operate> menu of the category view (Figure 4.10).

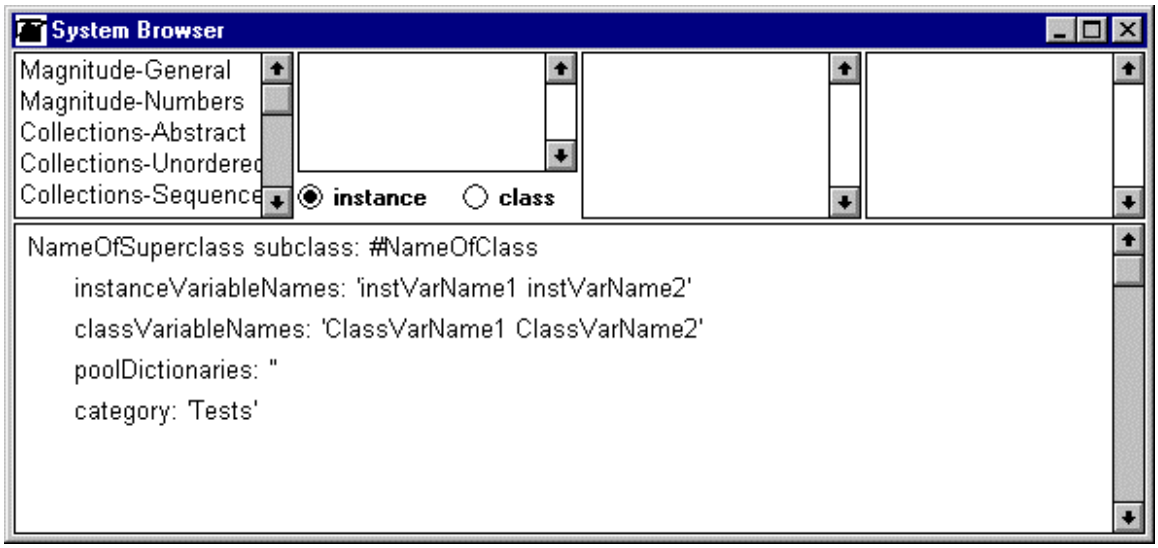

Figure 4.10. System Browser showing a class template.

The Browser now displays a template for creating a new class, actually a Smalltalk message whose execution will create the class. To complete the template, we must enter the name of the superclass (we will use Object because no related class is available), the name of the class (Name), names of instance variables (as decided above), and names of class variables (none) and poolDictionaries (none). Finally, click accept in the <operate> menu of the code and unless you made a typing mistake, the new class is added to the library. The resulting browser showing the completed template and the new class is as in Figure 4.11.

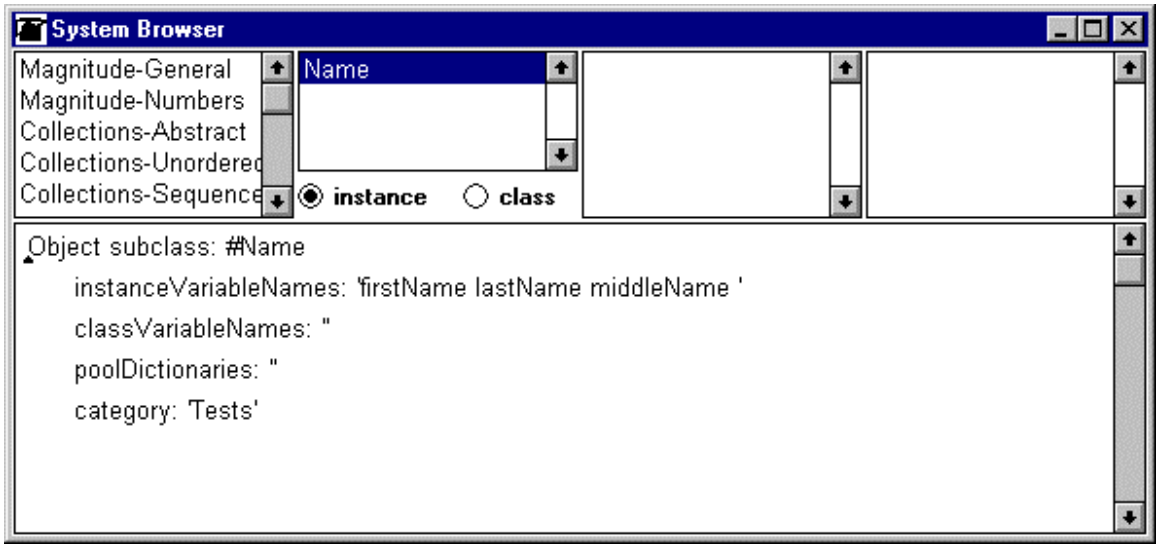

Figure 4.11. Browser after adding class Name.

The next step is to add class comment and to do this, click comment in the class view's <operate> menu. The bottom view of the browser changes and displays the following text:

This class has not yet been commented. The comment should state the purpose of the class, what messages are subclassResponsibility, and the type and purpose of each instance and class variable. The comment should also explain any unobvious aspects of the implementation.

Select the whole text and replace it with a comment. There are two styles for writing the comment. One is very antropomorphic and uses the first person as in 'I represent a name ..'. Its advantage is that it makes you think in terms of the class which always helps. The other style is less personal and uses the third person as in 'This class represents a name …'. This style seems less extravagant. Choose the style that you like better and use it consistently. We will use the first style because it will force us to think as if we were the class which seems a good idea at this point of the book. Following the template, we write

I represent a personal name consisting of a first name, a middle name, and a last name.

Instance variables

firstName	<String>	first name
middleName	<String>	middle name
lastName	<String>	last name

The comment seems almost unnecessary because it seems very obvious but it is an essential practice to use class comments. As an example, the indication that the values of variables should be String objects is important.
At this point, open a workspace and execute

Name new

with *inspect*. Method new is inherited by all classes from class Behavior and it creates a new instance of the class with uninitialized instance variables. Obviously, this is not good enough and we must now create methods to assign values to these instance variables, and to access the values once the object exists (because the values of instance and class variables can only be accessed by instance or class messages). Such messages are called *accessing* messages and they come in two varieties which are often called *get messages*, and *set messages*. Get messages return the value of the variable, set messages set its value to a new object. We will thus define accessing methods for all our variables and put them into a protocol called accessing. (The most common kinds of protocols have established names which are listed in Appendix 4.) To do this, use command *add …* in the <operate> menu in the protocol view of the browser. You will get the template in Figure 4.12. The template explains the basic structure of a method (as we already explained) and to create a method, simply select all the text and start typing the definition of the method.

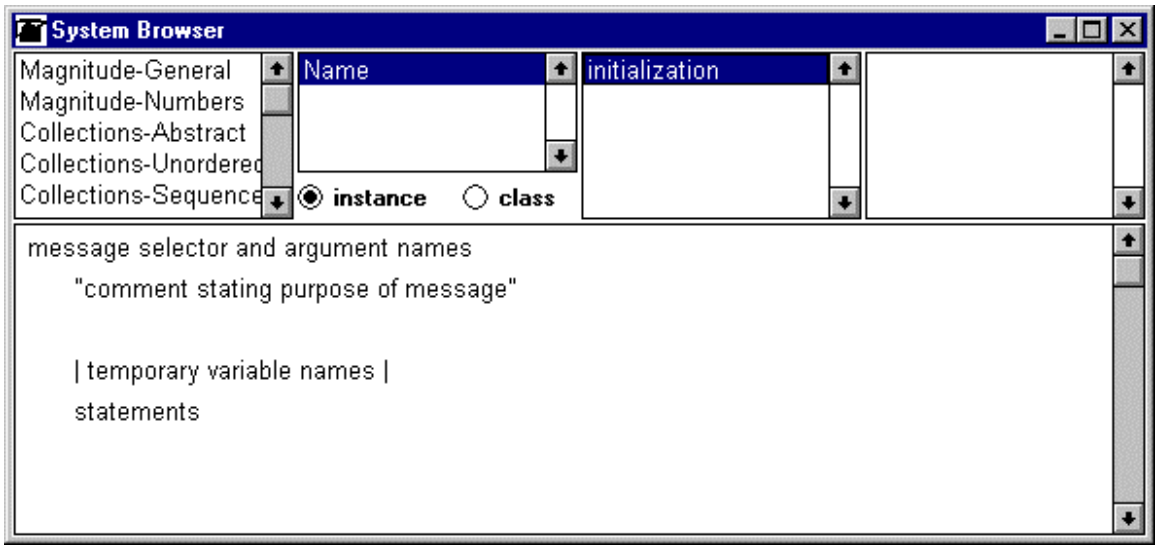

Figure 4.12. Browser with method template.

Let's start with the get method for the firstName instance variable. A get message simply returns the value of a variable and it is standard practice that it is named with the name of the instance variable. The definition is thus

```
firstName
        ^ firstName
```

Accessing methods are so common and so simple that programmers don't normally comment them.

After typing the method, select the *format* command in the <operate> menu of the code view and the code will be automatically formatted. If you made syntax mistakes, you will get a warning and you will have to correct your mistake. After this, execute the accept method which will compile the code and add the method to the library (Figure 4.13).

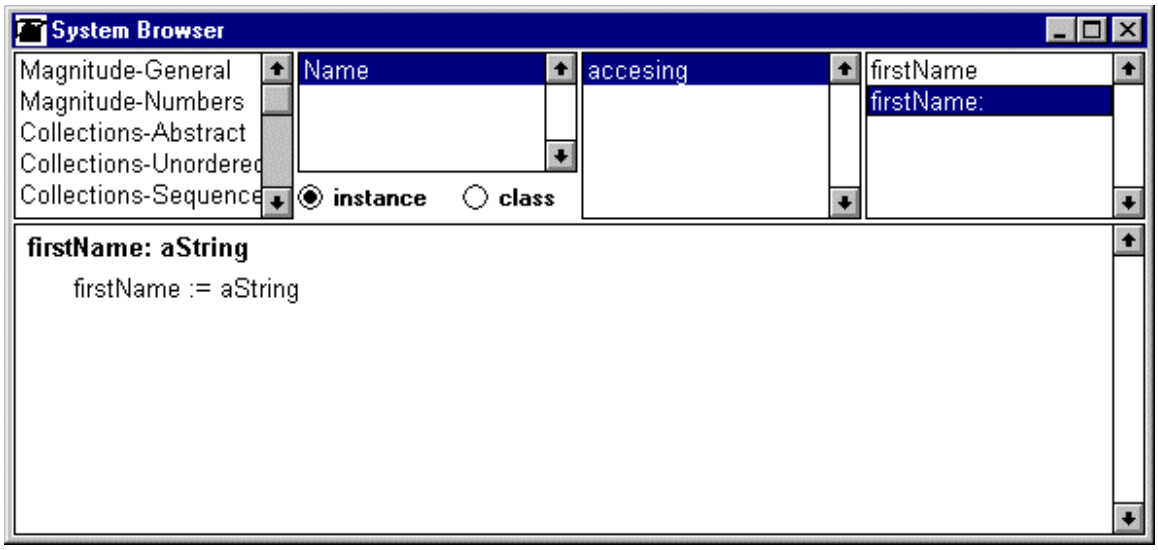

Figure 4.13. Browser showing the new accessing method.

We will now write the corresponding set method and leave it to you to define the remaining get methods. A set method simply assigns a new value to an instance variable and the new value is supplied as the argument. Our set method is thus a keyword method:

```
firstName: aString
        firstName := aString
```

We leave it to you again to define the remaining set methods and proceed to a little test. We should now be able to create a new Name object and assign values to its instance variables. As an example,

```
Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith'
```

should create a Name corresponding to John Mathew Smith. Assuming that you created the remaining set accessors, type this code into a workspace and execute with inspect. When you examine the instance variables, you will find that they indeed have the desired values.
As our next step, we will test our get methods. As an example,

```
(Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith') firstName
```

should return 'John'. Type this expression into a workspace and execute with *print it* to see that it indeed does.

Now that we think about it, it seems that our approach could cause a problem. A programmer might, for example, create a Name object and give it a first and last name but no middle name. If a program then tried to print the name, it would fail on the middle name because its value would be nil and nil does not understand String messages. It will thus be a good idea to modify the instance creation message so that it initializes all instance variable to empty strings with no characters in them. This is, of course, rather useless but at least it will not cause a crash if a variable is left unassigned.

There are several possible solutions to our problem and the simplest one is to define a new initialization method that forces the user to enter some strings for each of the instance variables as in

```
Name first: 'John' middle: 'Mathew' last: 'Smith'
```

This new message is obviously a class message because the receiver is the class which creates the object. We will thus define it on the class side and put it into a new protocol called instance creation . Its execution consists of creating an instance using new, and initializing it with accessing messages as we did above. The definition is thus

```
first: firstString middle: middleString last: lastString
"Create and return an initialized instance."
        ^(self new) firstName: firstString; middleName: middleString; lastName: lastString
```

A few notes are in order:

1. If we did not include the ^ symbol, the method would return the receiver (the *class* Name) rather than the created *instance*. This is a typical omission that frequently creates problems.
2. We could have used Name instead of self with the same result because the receiver of this message (object self) is class Name. However, this could create an undesirable effect if we subclassed Name and the style shown above is thus the preferred one.
3. We could have left out the brackets around self new and get the same result: Message firstName: message would go to the new instance of Name, and the remaining two messages would go to the receiver of firstName:, in other words the new instance of Name (now modified by message firstName:).

Class Name is now complete and we leave it to you to test that our new instance creation message indeed works as desired.

Example 2. Test of asUppercase

We have already mentioned that all code must be carefully tested. To test a new class, we usually write methods that test the essential methods of the class and add them to the class protocol. Even better, we could write a special class containing the tests. In this example, we will use the first approach and write a simple method to test the asUppercase method.

Although there is no general rule for writing test methods, the principle is that we find some situations where the tested method should do useful work, and some situations where it should not do anything and test both. We must test both typical and 'boundary' situations.

In our case, asUppercase should convert all lowercase letters to uppercase but it should not affect any other characters. In particular, it should not change any uppercase letters, digits, punctuation symbols, and spaces. The method should work on strings consisting of lowercase letters only, as well as strings containing lowercase letters and other characters.

Our test method will work as follows: For each tested situation, we will construct a string, send it the asUppercase message, and compare the returned and the expected result. We will then write a message reporting the success or failure of the test to the Transcript. As an example, the first test might be

```
'abcd' asUppercase = 'ABCD'
        ifTrue: [Transcript show: 'test 1 passed']
        ifFalse: [Transcript show: 'test 1 failed']
```

We will add the test method to class CharacterArray because that's where asUppercase is defined, and define it in a *class* protocol. Since no suitable class protocol exists, we will create a new one called testing. The definition of the method will be as follows:

```
testConversion
        "Test asUppercase conversion on several selected receivers."

        Transcript clear.
        'abcd' asUppercase = 'ABCD'
                ifTrue: [Transcript show: 'test 1 passed']
                ifFalse: [Transcript show: 'test 1 failed'].
        Transcript cr.
        'ABCD' asUppercase = 'ABCD'
                ifTrue: [Transcript show: 'test 2 passed']
                ifFalse: [Transcript show: 'test 2 failed'].
        Transcript cr.
        'abCd' asUppercase = 'ABCD'
                ifTrue: [Transcript show: 'test 3 passed']
                ifFalse: [Transcript show: 'test 3 failed'].
        Transcript cr.
        '123' asUppercase = '123'
                ifTrue: [Transcript show: 'test 4 passed']
                ifFalse: [Transcript show: 'test 4 failed'].
        Transcript cr.
        'ab12?Cd' asUppercase = 'AB12?CD'
                ifTrue: [Transcript show: 'test 5 passed']
                ifFalse: [Transcript show: 'test 5 failed']
```

To add the method to the library, open a System Browser on class CharacterArray, create a testing protocol on the class side using the technique explained in Example 1, and enter, format, and accept the new method.

Once the method is compiled, we must test that it indeed works. To perform the test, send the test message to class CharacterArray (or to any of its subclasses) as in

```
CharacterArray testConversion
```

The message prints confirmations that all tests succeeded into the Transcript view.

Assume now that this is a convenience method and that we don't want to clog the library with it. We will thus save its code in a file and remove the method from the library. To do this, open the <operate> menu in the method view of the System Browser and select command *file out as* ... (Figure 4.14).

Figure 4.14. <operate> menu command for filing out a method. A similar menu is available in other System Browser views to file out a whole protocol, a whole class, or a whole category of classes.

The command opens a dialog requesting the name of the file as in Figure 4.15. Accept the proposed name or enter another name and possibly a new directory path, and click *OK*. If you did not specify a new directory path, the file (a *fileout* of the code) will be stored in the image sub-directory of your VisualWorks directory.

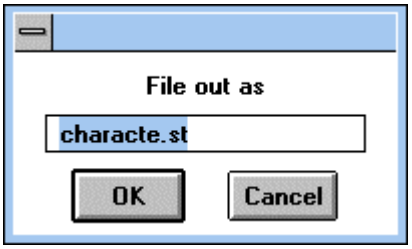

Figure 4.15. The fileout dialog with the default filename. The default directory is image.

Now that we saved the method, we can delete it and its protocol from the library using the *remove* command in the <operate> menu of the protocol view. If you ever want to restore the protocol and the method (to test asUppercase after some changes), open the File Editor on the file (Figure 4.16) and file the code in using the *file in* command of its <operate> menu. This reads the file and restores its contents (method, protocol, class, or category) in the library. This happens because fileout files use a special format and as they are filed in, the compiler automatically translates them back into the library.

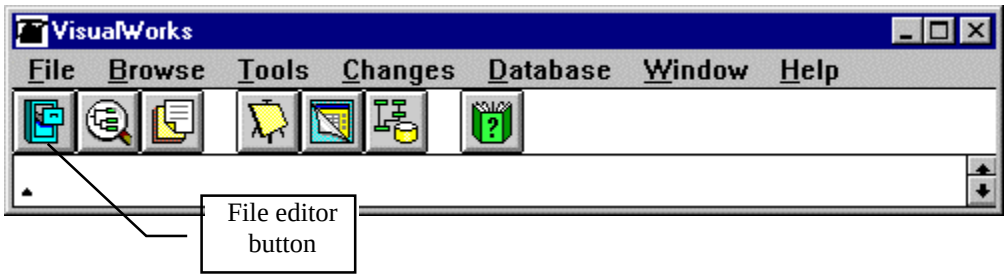

Figure 4.16. To open a File Editor, use the leftmost button or the *File Editor* command in *Tools*.

Main lessons learned:

- Use the System Browser to create new categories, classes, protocols, and methods.
- Testing is usually implemented by adding test methods to the class being tested or by creating a special test class.
- Collect test methods in a testing protocol on the class side.
- Remove test methods from the library when not needed but save them in a file first.
- To save a method, a protocol, a class, or a category in a file, file it out. To restore it, file it in.
- Fileout files use a special format and the compiler restores the contents into the library during fill-in.

Exercises

1. Implement the examples in this section.
2. It would seem that instead of typing
   (Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith') firstName
   we could have used
   Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith' firstName
   with exactly the same result. Is that correct?
3. Protocol converting in CharacterArray contains several other conversion methods. Extend our test method to test methods asLowercase and asNumber.

4. Write and execute a test method to test some of the methods in CharacterArray protocol *comparing*. Include tests of methods < and > (these methods compare strings on the basis of their collating sequence - basically their alphabetical order) and match: which compares strings using wildcard characters (* matches any string, # matches any single character). The comment of match: includes examples demonstrating the purpose and use of the method. Including examples in method definition is a common practice.
5. Write and test method requestNumber: aString initialAnswer: aNumber. The method should work just like request:initialAnswer: but return a number. (Hint: Use the principle that we used in Section 4.4.) What is the best class for the method?
6. Define and test method cubed to calculate the third power of a number. Use squared as a model.
7. Define class Address with instance variables street, number, city, postalCode. The values of all instance variables are strings. Define all necessary methods and test them.
8. Define class RentalProperty with instance variables address (an instance of Address from Exercise 7) and numberOfApartments (an integer). Define all necessary methods and test them.

4.7 Logic operations

A test condition often has several components. As an example, when we want to check whether a number is greater than 5 and divisible by 7, we must combine two tests into one. Operations combining true and false objects are called *logic operations* and the Smalltalk library provides several methods to implement them.

The most common logic operations are called *not*, *and*, and *or*. Although the names are derived from English and make a lot of sense, their interpretation could be ambiguous and the definition is thus better expressed by tables. Because the tables deal with true/false values, they are usually called *truth tables*. The truth tables of *not*, *and*, and *or* are given in Table 4.1 and you can check that their meaning corresponds to the meaning of the words *not*, *and*, and *or* in everyday communication. As an example, if *a* is true, *not a* is false, and so on.

a	not a
false	true
true	false

a	b	a and b
false	false	false
false	true	false
true	false	false
true	true	true

a	b	a or b
false	false	false
false	true	true
true	false	true
true	true	true

Table 4.1. Truth tables of (left to right) *not a*, *a and b (conjunction)*, *a or b (disjunction)*.

In Smalltalk, not, and, and or are defined in classes Boolean, True, and False and used as follows:

```
aBoolean not
aBoolean and: [“statements that evaluate to true or false“]
aBoolean or: [“statements that evaluate to true or false“]
```

They all return a Boolean result and we will now illustrate them on several examples.

Example 1. Test whether an income is taxable

Problem: Write an expression that tests whether a tax payer’s income is taxable or not. The rules are as follows: Income is taxable if the person is up to 60 years old and the income is greater than $25,000, or the person is over 60 and the income is greater than $30,000.

One possible solution is to evaluate the two rules separately, store the results in variables, and combine the variables. This can be expressed as follows:

```
| age income rule1 rule2 |
age := (Dialog request: 'Enter your age' initialAnswer: '') asNumber.
income := (Dialog request: 'Enter your income' initialAnswer: '') asNumber.
```

```
rule1 := age <= 60 and: [income > 25000].
rule2 := age > 60 and: [income > 30000].
(rule1 or: [rule2])
        ifTrue: [Dialog warn: 'Tax is payable']
        ifFalse: [Dialog warn: 'No tax payable']
```

Another possibility is to eliminate the rule variables and combine all the rules into a single calculation as follows:

```
| age income |
age := (Dialog request: 'Enter your age' initialAnswer: '') asNumber.
income := (Dialog request: 'Enter your income' initialAnswer: '') asNumber.
((age <= 60 and: [income > 25000]) or:
        [age > 60 and: [income > 30000]])
                ifTrue: [Dialog warn: 'Tax is payable']
                ifFalse: [Dialog warn: 'No tax payable']
```

This code has exactly the same effect but it is harder to read and more error prone. We prefer the first solution.

Example 2. Safeguarding against an illegal message

The fact that the argument of and: and or: is a block is important because a block is not just a group of statements but rather a group of statements whose evaluation is deferred until explicitly requested by a value message. In the case of and: and or: this allows us to safeguard against an illegal message. As an example, consider the following fragment:

```
…
(x >= 0 and: [x sqrt <= 5])
ifTrue: [Dialog warn: 'Argument ', x printString, ' is legal']
        ifFalse:   [Dialog warn: 'The argument is illegal]
…
```

This code fragment works for any value of x because it evaluates the block argument of and: only if the first test succeeds and defers (and never executes) it if the test fails. If this were not so and if the value of x was negative, the program would crash when attempting to execute the statement in the block argument because negative numbers crash on the sqrt message.

Example 3: Use of and: in the definition of between:and:

Method between:and: returns true if the receiver is greater or equal to the first argument and less or equal to the second argument:

```
3 between: 1 and: 5          "Returns true."
```

whereas

```
3 between: 10 and: 50        "Returns false."
```

Its definition in class Magnitude is as follows:

```
between: min and: max
        ^(self >= min) and: [self <= max]
```

The method takes the receiver and sends it the message >=, getting a true or false result. If the result is false, it does not evaluate the block because the result must be false (see Table 4.1). If the result is true, it sends and: with

```
[self <= max]
```

to the true object. This returns true or false. The Boolean result is then returned. Note that all subclasses of Magnitude including Date, Time, Character, and all number classes inherit this method.

Example 4: A ticketing program

Problem: A computer program issues tickets in a museum and calculates the price on the following rule: A ticket is $1 on weekdays between 9 a.m. and 12 a.m., on Saturday after 12:00, and any time on Sunday. The rate is $2 at all other times.

Solution: The essence of the solution is as follows:

1. Get current time.
2. Get today's day of the week.
3. Use the time and date to determine which rate applies.

To write the program, we need to know how to find the current time and day of the week. We suspect that there are classes and messages to do this and we indeed find classes Time and Date.

When we check the instance methods of class Time, we don't find anything that appears suitable for our purpose. We thus open class protocols and find method called now in the instance creation protocol. Its comment says "Answer a Time representing the time right now--this is a 24 hour clock". To test this method, we execute

```
Time now
```

with *inspect*, and get a Time object with components hours, minutes, and seconds. Hours are counted from 0 to 23, minutes and seconds are counted from 0 to 59. Our next question is how to extract the hours component. Time now returns an instance of Time and we thus check instance methods of Time at the instance methods side; in the accessing protocol, we find method hours. When we execute

```
Time now hours
```

with *print it*, we get just what we wanted.

Now that we know how to get time, we search class Date. We expect that the method returning today's date will be a class method, probably in the instance creation protocol and we indeed find a method called today which seems to do the job. Executing

```
Date today
```

with *inspect*, returns a Date object with two instance variables called days and year. We are interested in days of the week but is that what does days mean? When we open the comment on Date we find that days is the number of days since the start of the year. Is there some way to convert this number into the day of the week? If there is, it must be an instance method because Date today is an instance of Date. We thus look under the instance methods of Date and find method weekday that seems to do exactly what we need. When we try

```
Date today weekday
```

it returns a string such as 'Thursday'. We now have all the ingredients and we can write the code which uses heavily the or: method:

```
| dayToday price |
"Get time and day now."
timeNow := Time now hours.
dayToday := Date today weekday.
"Find which rate applies."
(((dayToday = 'Sunday')
        or: [dayToday = 'Saturday' and: [timeNow >= 12]])
        or: [timeNow between: 9 and: 12])
                ifTrue: [price:= 1]
                ifFalse: [price:= 2].
Transcript clear; show: 'price is ', price printString
```

Example 5: Multiple logic operations

Problem: Get user's gender, age, and weight. If the user is male, over 50 years of age, and weighs 150 kg or more, display a warning saying 'You should take a look at your diet'.

Solution: To obtain the information, we will use multiple choice dialogs. These dialogs are easy to construct - the one in Figure 4.17 was obtained with the following code:

```
Dialog    choose: 'What is your gender?'                  "Prompt message."
          labels: (Array with: 'female' with: 'male' )    "Labels on buttons."
          values: (Array with: 'female' with: 'male' )    "Objects returned when a button is clicked."
          default: nil                                    "Specifies initially selected button."
```

The code uses some concepts that we have not yet covered but the pattern is obvious and we will use it without further questions.

Figure 4.17. A multiple choice dialog for Example 3.

The logic of the problem requires that we combine two and: messages and to do this, we must nest the second and: into the block argument of the first and:. The complete solution is as follows:

```
| gender overFifty largeWeight |
"Get the information."
gender := Dialog  choose: 'What is your gender?'
                  labels: (Array with: 'female' with: 'male' )
                  values: (Array with: 'female' with: 'male' )
                  default: nil.
overFifty := (Dialog confirm: 'Are you at least 50?').
largeWeight:= Dialog confirm: 'Is your weight 150 kg or more?' .
"Evaluate collected information and output warning if appropriate."
((gender = 'male') and: [overFifty and: [largeWeight]])
          ifTrue: [Dialog warn: 'You should take a look at your diet.']
```

How are logic operations implemented?

You might expect that the definition of and: takes the receiver and the argument and uses the truth table to decide whether to return true or false. In fact, it does not because there is a better way. In class False, and: simply returns false (the receiver)

```
and: alternativeBlock
"Don't bother evaluating the block - according to the truth table, the result must be false."
          ^self
```

because the truth table of and shows that when the receiver is false, the result must also be false.

We have already mentioned that the fact that this definition eliminates evaluation of the block argument may be used as a safeguard. It can also save a substantial amount of time because the calculation could be very time consuming as in

```
30 between: 5 factorial and: [30000 factorial / 256000.13 squared]
```

and is not needed if we only want to know whether the result is true or false. This approach is known as *non-evaluating conjunction* or *short-circuited conjunction* because it eliminates evaluation of the argument

block when it is not necessary to obtain the true or false result. The or: message is implemented in a similar way.

The definition of and: in class True is also based on the truth table. When you check what is the value of and when the receiver is true, you will find that the result is always the same as the argument. The definition in class True thus simply evaluates the argument block and returns it:

```
and: alternativeBlock
"Evaluate the block because the result might be true or false. The value is the same as the block's value."
    ^alternativeBlock value
```

Fully evaluating *and* and *or*

Non-evaluating *and* and *or* may not be acceptable when the argument block does something that must not be skipped. As an example, assume that you want to draw two pictures and check whether both fit within some rectangle R. A possible approach is

"Draw the first picture and calculate its bounding box (the rectangle containing it). Check whether the bounding box is included in rectangle R."
and
"Draw the second picture and calculate its bounding box. Check whether the bounding box is included in rectangle R."
"Return the result (true or false."

If we implemented this logic using and: and if the first bounding box was not in rectangle R, the block that draws the second picture would not be evaluated and the second picture would not be drawn. This may not be what we wanted because the *side-effect* of the block (the drawing of the rectangle) may always be desired. For situations such as these, Smalltalk provides *fully-evaluating and* and *or* denoted & and |. Fully-evaluating logic always evaluates both arguments even when this is not required by the truth table. We will see below that both of these messages take a Boolean rather than a block as their argument.

If we have a choice, non-evaluating logic is better because it may be faster (when the evaluation of the argument is not required, it is skipped) and safer. It also provides the option of exiting from a method by using a ^ return operator in the block. Fortunately, we can always eliminate fully evaluating logic because there is always a way to solve a problem without side-effects. As an example, a better way to solve the above problem is as follows:

"Draw the first picture and calculate its bounding box box1."
"Draw the second picture and calculate its bounding box box2."
"Check that box1 and box2 are within R using *non-evaluating* conjunction."

Stylistically, fully-evaluating logic also has its advantage – it can make the code easier to read. As an example, the nested conditions in Example 3

```
((gender = 'male') and: [overFifty and: [overweight]])
    ifTrue: [Dialog warn: 'You should take a look at your diet.']
```

could be implemented in a more readable fashion with & as follows:

```
(gender = 'male') & overFifty & overweight
    ifTrue: [Dialog warn: 'You should take a look at your diet.']
```

with the same effect. If the run-time penalty is acceptable, there is nothing wrong with this approach.

Main lessons learned:

- Logic operations such as *not, and*, and *or* are defined by truth tables derived from natural language.
- Logic *and* and *or* can be implemented as fully-evaluating or non-evaluating operations. Fully evaluating logic always evaluates both the receiver and the argument, non-evaluating logic evaluates the argument only if it is necessary.
- Non-evaluating (shortcut) logic with and: and or: speeds up execution and provides a means to protect against attempts to execute illegal messages.
- Non-evaluating logic *must* be used if the evaluation of the receiver is also a test whether the block argument should or should not be evaluated.
- Blocks represent deferred evaluation which means that the statements inside a block are ignored until their execution is explicitly requested.
- Smalltalk library contains built-in classes and methods to do almost anything. To learn Smalltalk, learn how to search the library.

Exercises

1. Write programs to solve the following problems:
   a. Tax calculation: Tax is 0 if the person's income is less than $10,000, or if the age is less than 18 years or more than 65 years. The tax is 10% of total income for all other income earners.
   b. Student registration fee is $5,000 unless the student's grade average is between 80 and 90 (inclusive) and the student lives in a residence (registration $4,000), or the grade average is between 91 and 100 (registration fee $2,000).
2. If you did the previous exercise, you found that when we need to check multiple conditions, the expression can get quite ugly and confusing because of the many brackets. Define two new methods and: block1 and: block2, and or: block1 or: block2 that allow two conditions to be packed into one message and make the code somewhat more compact and readable. Use the method to re-implement Exercise 1.
3. Correct the bracketing of the following expressions to make it syntactically correct. Assume that we are not using the methods from Exercise 2.
   a. (square1 intersects: square2) and: [square1 includes: point1 and: square2 includes: point2]
   b. x between: 3 and: y or: [(x > 1000 or: (x < -1000)]
4. Read and explain the definitions of or:.
5. Read and explain definitions of & and | - fully-evaluating conjunction and disjunction.
6. How many times are and:, or:, | and & used in the base library?
7. List five most useful methods of Date and Time.
8. Test the ticketing program with suitable combinations of days and times to make sure that its logic is correct. (Hint: Since we are testing the logic, it does not matter how the Date and Time objects are created. Use suitable alternative creation messages.)
9. Select a method in protocol testing in class Rectangle and explain how it works.
10. Predict the result of the evaluation of the following expressions, test, and explain:
    a. (3 < 5) and: [6 squared]
    b. (30 < 5) and: [6 squared]
    c. (3 < 5) & [6 squared]
    d. (30 < 5) & [6 squared]
    e. $a between: 7 and: $z

4.8 Exclusive or, equality, and equivalence

Besides *not*, *and*, and *or*, another useful logic function is *exclusive or* (usually abbreviated as *xor*). Its definition is as follows:

x xor y

is true if *x* and *y* have different Boolean values, and false otherwise. In Smalltalk, *xor* is implemented as a Boolean keyword message xor: with one *Boolean* argument

```
aBoolean xor: anotherBoolean
```

As an example of the meaning of *xor*, assume that a restaurant customer ordering from a fixed menu must choose either a cake or an ice cream but not both. An expression to check that an order follows this rule could be written as

```
legalOrder := ordersCake xor: ordersIceCream
```

where ordersCake and ordersIceCream are true or false. If the expression returns true, the order is OK, otherwise, something is wrong.

Implementing xor: requires only checking whether the receiver and the argument are the same objects and then inverting the result using logical *not*. There is no need to have one definition for True and another for False, and xor: is thus defined in class Boolean as

```
xor: aBoolean
        ^(self == aBoolean) not      "The == message tests equivalence."
```

Classes True and False inherit this definition.

The definition of xor: introduces the notion of *equivalence*, an important new concept implemented by the == binary message. *Objects x and y are equivalent if they are one and the same object*, in other words, if they are identical. Compare this with *equality*, a relation implemented with the = message: *Two objects are equal if their values are equal in some well-defined sense*. Obviously, two equivalent object are equal (because they are the same), but the opposite may not be true as in the following example: When VisualWorks Smalltalk executes

```
| x y |
x := 'John Smith'.
y := 'John Smith'
```

the first assignment statement creates an internal representation of the string 'John Smith' and stores it in memory. The second statement again creates an internal representation of the string 'John Smith' and stores it in memory. We now have two different representations of 'John Smith' (Figure 4.18) which are equal in the sense of string equality (the corresponding characters of the two strings are the same) - but *not* equivalent because each has its own identity, its own bit pattern stored in its own place in memory.

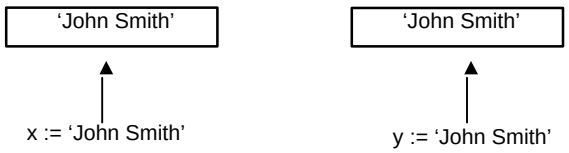

Figure 4.18. Two strings may be equal while not being equivalent. x=y is true but x==y is false.

On the other hand, x and y in

```
| x y |
x := 'John Smith'.
y := x      "y is bound to the same object as x."
```

are not only equal but also equivalent because both y and x are bound to the same object (Figure 4.19).

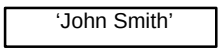

Figure 4.19. y := x := 'John Smith' binds x and y to the same object and both x=y and x==y are true.

As another example, consider that class True and False each have only one instance. As a consequence, equivalence and equality of Boolean objects mean the same thing.

The check of equivalence is faster because it only compares the memory addresses of the internal representation of the two objects. If the addresses are the same, the objects are equivalent; they are one and the same object. Checking whether two objects are equal may take much longer. As an example, to decide whether two non-equivalent strings are equal, we would probably first check whether they have the same size (if not, they cannot be equal) and if they do, we would have to check the individual character pairs one after another. Imagine how long this will take if the two strings are, for example, two versions of this book!

If the test for equivalence is so much simpler, why do we need equality? There are at least two reasons. One is that restricting equality to equivalence would sometimes be very inefficient. Consider again strings. If two strings that contain the same characters had to share one memory representation[2], then every time that we change a string, we would have to check whether we already have a string with the same characters. If we do, we must discard one and make all reference to it point to the already existing instance. This would be extremely inefficient.

Another reason why we need both equality and equivalence is that two objects are sometimes 'equal' even though they are not quite the same. As an example, two $100 bills are interchangeable in terms of their value even though they are not one and the same object. In fact, a 'currency object' consisting of two $50 bills and is also interchangeable with a $100 currency object.

As another example, consider filenames. On some platforms, filenames are case insensitive, and strings such as 'filename' and 'FileName' are then two different versions of the name of the same file. Consequently,

```
'abc' asFilename = 'ABC' asFilename   "true on platforms with case insensitive filenames."
```

On case insensitive platforms, we could thus define = for filenames as

```
= aFilename
^ self string asLowercase = aFilenameString string asLowercase
```

where we assumed that the name of a filename is accessed by message string.

When should we use equivalence and when should we use equality? The answer is that you should *always use equality* unless you want to check that two references refer to the same object. In those situations in which equivalence and equality mean the same thing, they are defined to mean the same thing and there is no loss of speed in using equality. This definition is inherited from Object where

```
= anObject

        ^self == anObject
```

It is very important to note that when = is redefined, the hash method must also be redefined. We will see what hash is and why it is so closely related to equality later.

[2] For special needs, there is a subclass of String called Symbol in which two identical values are stored only once and share the same location

Main lessons learned:

- Exclusive *or* of two Boolean arguments returns true if and only if the arguments are different.
- Two objects are equivalent (message ==) if they are one and the same object. Equality (message =) means that two objects are in some sense interchangeable.
- The definitions of = and == inherited from Object give the same result for equality and equivalence. Many classes, however, redefine equality to satisfy their special meaning of 'interchangeable'.
- Testing for equivalence is very simple and fast, testing for equality may take much longer.
- Equivalence is stronger than equality: If two objects are equivalent, they are also equal, but two equal objects may not be equivalent.
- Always use equality unless you need to know that two expressions refer to the same object.

Exercises

1. Draw the truth table of exclusive *or*. Can *xor* be short-circuited? If not, how does this relate to the fact that the argument of xor: is a Boolean and not a BlockClosure?
2. List all combinations for which the ice cream - cake order test returns false.
3. Implement the following problems:
   a. Female patients who have a sore throat and a headache, and male patients who have a sore throat but not a headache get medication M1; other patients don't get any medication. Write a code fragment to request gender and the medical problem and advise on the medication.
   b. A qualified programmer will be hired if she wants to work in Santa Barbara and accepts a salary below $130,000, or if she requires $145,000 but accepts a location not in Santa Barbara. The program obtains the information and reports whether the candidate is acceptable.
4. Study the definition of xor: and comment on its consistency with the definition of *xor*. Are there any situations in which the definition will not produce the expected result?
5. Define equality of complex numbers with real and imaginary part. Note that it must be possible to compare complex numbers with other kinds of numbers.
6. Give your own example of a situation in which two objects should be considered equal even though all their components are not the same.
7. Count how many classes have their own definitions of = and explain two of them in detail.

4.9 Use of Booleans to repeat a block of statements

Almost all programs require repetition of a block of actions until some condition succeeds or fails. As an example, a search for a book in a library catalog must search catalog entries one after another until the desired entry is found or the search reaches the end of the catalog.

The process of repeating a block of actions until some condition is satisfied or fails is called *iteration* or *enumeration* and Smalltalk contains several methods to implement it. In the rest of this chapter, we will look at the most general ones and leave the more specialized for later.

Example 1: Print all positive integers whose factorial is not bigger than 15,765
Solution: The obvious solution is as follows:

1. Define a variable called, for example, number to hold the current value of the value over which we are iterating. Initialize number to 1.
2. Check whether number factorial is less or equal 15765. If yes, print the number, increment it by 1, and repeat this step, otherwise stop.

This algorithm can be implemented as follows:

```
| number |
number := 1.
```

```
Transcript clear.
[number factorial <= 15765]          "This is the condition for doing another iteration"
        whileTrue: [Transcript show: number printString; cr. number := number + 1]
```

In this example, iteration is performed with the whileTrue: message. Its general form is

```
conditionBlock whileTrue: iterationBlock
```

and it works as follows (Figure 4.20): The conditionBlock is evaluated; if the result is true, the iterationBlock is evaluated and conditionBlock is re-evaluated. Iteration continues until evaluation of conditionBlock returns false. Note that conditionBlock must always evaluate to true or false; if it does not, the message fails and opens an Exception Window.

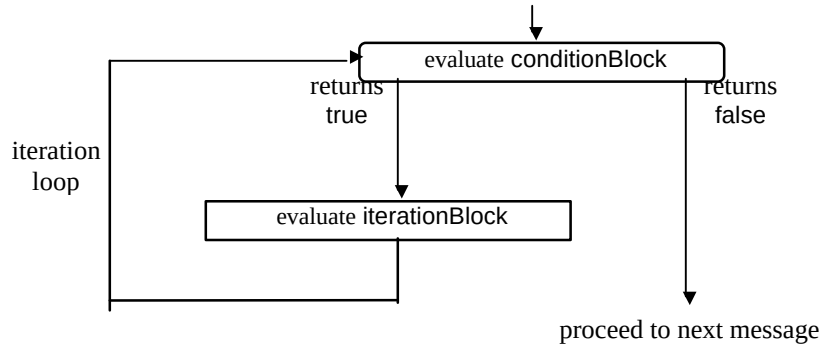

Figure 4.20. Evaluation of conditionBlock whileTrue: argumentBlock is based on looping.

The whileTrue: message is very useful but it must be used very cautiously. The danger is that if the receiver block never returns false, the program will continue looping around and never stop, a situation known as an *infinite loop*. Our program, for example, would get into an infinite loop if we forgot to increment the value of number in the iteration block.

In spite of all possible caution, you will undoubtedly eventually write a program that does get into an infinite loop and something that should take milliseconds will be running, and running, and running. If this happens, interrupt execution by pressing the <*Ctrl*> and <*c*> keys simultaneously. Smalltalk will open an Exception Window as if you halted the program with self halt and you can correct the program using the debugger or terminate execution.

Example 2: A simple shopping calculator

Problem: As a part of a Smalltalk application running a portable shopping calculator, we need a function that lets the user enter prices of grocery items, print them, and calculate and display a running total. When the total exceeds a preset limit, the program displays the total and displays a warning message.

Solution: The real application will require several classes and a graphical user interface (GUI) but since we are not in a state to implement it yet, we will test the general idea on a code fragment and a specific limit value, and display the required data in the transcript. The algorithm is as follows:

1. Initialize variable total to 0 and clear the Transcript.
2. While total <= 100 is true, do the following:
   get price of the next item
   display price in Transcript
   update total
3. Display warning window and print total in Transcript.

This algorithm can be easily converted into the following program:

```
| price total |
```

```
“Initialize total and clear Transcript.”
total := 0.
Transcript clear.
“Keep gathering item prices and displaying them until the total exceeds 100.”
[total <= 100]
        whileTrue:
                [price := Dialog request: ‘Enter next price’ initialAnswer: ‘’.
                Transcript show: price; cr.
                total := total + price asNumber].
“Open warning window, print result, and stop.”
Dialog warn: ‘The total has exceeded 100, end of input’.
Transcript show: ‘The total is now ‘, total printString
```

The whileFalse: message

Iteration is also available in the form whileFalse: which repeats execution as long as the receiver is false. You can always switch between whileTrue: and whileFalse: and the better choice is the one that makes more sense in the current context. With whileFalse:, the previous program could be rewritten as

```
| price total |
“Initialize total and clear Transcript.”
total := 0.
Transcript clear.
“Gather item prices, display them, calculate running total.”
[total > 100]                   “Inverted condition.”
        whileFalse:             “Instead of whileTrue:”
                [price := Dialog request: ‘Enter next price’ initialAnswer: ‘’.
                Transcript show: price; cr.
                total := total + price asNumber].
“Open warning window, display result, and stop.”
Dialog warn: ‘The total has exceeded 100, end of input’.
Transcript show: [The total is now ‘, total printString, ‘ - more than 100’
```

The definition of whileTrue: is based on recursion

How is the whileTrue: method defined? Since the receiver is a block, its definition is in class BlockClosure and the definition is as follows:

```
whileTrue: aBlock
        ^self value ifTrue: [aBlock value.
                             [self value] whileTrue: [aBlock value]]
```

This is a bit obscure and requires an explanation. The first line of the definition

```
^self value
```

evaluates the receiver, just as we would expect from Figure 4.20, and sends it the ifTrue: message.

If the returned value is false, the ifTrue: message has no effect and execution is finished. If the returned value is true, execution continues with

```
[aBlock value.
[self value] whileTrue: [aBlock value]]
```

Here aBlock value takes the argument block and evaluates it as in Figure 4.20, and execution proceeds to

```
[self value] whileTrue: [aBlock value]
```

which is equivalent to

```
[self value] whileTrue: aBlock
```

Since self value is the evaluated value of the original block, [self value] is the same as the original block self. As a consequence,

```
[self value] whileTrue: aBlock
```

is the same as

```
self whileTrue: aBlock
```

This agrees with Figure 4.20 which shows that when we finish one iteration, we do another one, just like the one before. In effect, when the condition of whileTrue: succeeds (returns true), the whileTrue: message is sent again, and again, and again, until the condition finally fails.

A message that sends itself is called *recursive* and the concept is related to recursive definitions which we have already encountered.

Although recursion looks like magic, it is very natural even in real life situations. Consider, as an example, a person asking how to find the railway station in a large city. If the railway station is far, the best answer might be something like 'Go to the third intersection and ask again.' This advice is recursive because it requires re-execution of the original request.

Recursion often considerably simplifies the solution of a problem but it is often very inefficient because each recursion step produces a new message send and this results in extra execution time and memory overhead. For this reason, Smalltalk tries not evaluate whileTrue: using recursion and the compiler converts whileTrue: messages to in-line code whenever it can. The in-line code - direct translation into a sequence of machine instructions - will work faster and requires less memory.

The situations under which the definition of whileTrue: is in-lined is described in the comment of whileTrue: which goes essentially as follows:

"*This method is in-lined if both the receiver and the argument are literal blocks*. In all other cases, the code above is run.
Note that the code above is defined recursively. *However,* to save time and memory each time this method is invoked recursively, we have used the '[...] whileTrue: [..]' form in the last line, rather than the more concise 'self whileTrue: aBlock'. Using literal blocks for both the receiver and the argument allows the compiler to inline #whileTrue:, which could not be done if we were to use 'self whileTrue: aBlock'."

The comment refers to the concept of a *literal block* which is a block of statements surrounded by square brackets such as

```
[Transcript clear]
```

In other words, a literal block is a block that is directly recognized by the compiler. On the other hand, the block stored in the aBlock argument of the definition of whileTrue: is not literal because the compiler cannot determine whether the value of aBlock is a block or not. In fact, if we wrote something like

```
[x < 3] whileTrue: Transcript clear
```

the argument would not be a block but the expression should at least start executing because the receiver (a ClcokClosure) understands the message.

The comment in the definition of whileTrue: means that if the argument of whileTrue: is a literal block, the compiler can recognize it as such and create in-line code. If the argument is not a literal block, the compiler cannot create in-line code because it does not have enough information. In this case, it will generate code to execute the definition (rather than creating in-line code as it would otherwise) by sending messages in the usual way. As an example, the whileTrue: message in

```
| block x |
x := 1.
```

```
block := [x := x + 1].
[x squared < 10] whileTrue: block.
Transcript show: (x - 1) printString
```

will be executed by message sends (executing the whole definition of whileTrue:) rather than in-lined because the argument is not a literal block. When you attempt to execute this example, Smalltalk will open the window in Figure 4.21 which warns you that the argument of whileTrue: is normally a literal block and asks you if you really intend to use a non-literal. In this case, a non-literal is exactly what we want so click *proceed* and the program executes as expected.

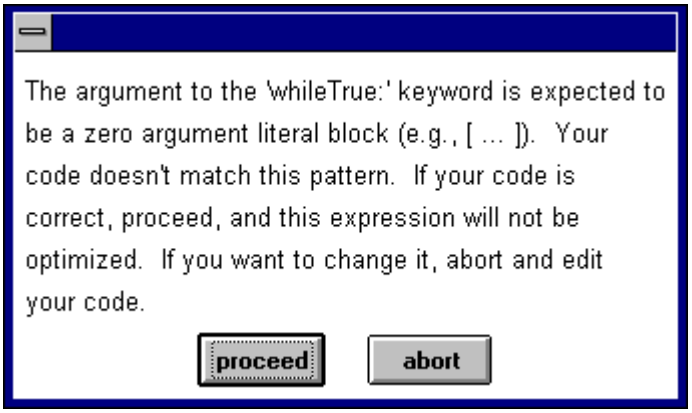

Figure 4.21. Warning produced when the argument of whileTrue: is not a literal block.

Besides the point that we have just explained, the comment has another even more obscure part that deals with the case when the argument is *not* a literal block. The essence is that if the compiler cannot in-line the original formulation, the recursive part is formulated so that the compiler *can* in-line the second execution and thus execute all iterations beyond the first one more efficiently. Consider the definition again:

```
whileTrue: aBlock
        ^self value
                ifTrue:
                        [aBlock value.
                        [self value] whileTrue: [aBlock value]]
```

If self or aBlock are not literal blocks, the message cannot be in-lined and the previously compiled definition executes. Now consider how this code has been compiled. Where the body contains a whileTrue: message:

```
[self value] whileTrue: [aBlock value]
```

the receiver and argument are both literal blocks and the compiled code is thus in-lined. Although the first iteration was thus executed less efficiently as a message, the remaining iterations are efficient because they use the in-lined code. Only the first iteration will thus be truly recursive, saving both execution time and memory space. This saving would not be possible if the formulation of the iteration was the more natural

```
self whileTrue: aBlock
```

because this code would again use a non-literal block.

Main lessons learned:

- Iteration is repeated execution of a block of statements. In its whileTrue: form, iteration stops when the receiver block returns false, in the whileFalse: form, it stops when the receiver block returns true.
- If the stopping condition is never satisfied, an infinite loop occurs. To interrupt an infinite loop, press <*Ctrl*> and <*c*> simultaneously.
- Recursion is the technique of defining a method in terms of itself.
- Recursion often leads to very simple solutions but it is usually inefficient in terms of speed and memory space.

Exercises

1. What is the value returned by the whileTrue: message?
2. Solve the following problems:
   a. Open a confirmer asking the user whether to stop or not. If the answer is yes, the program stops, otherwise the confirmer is displayed again.
   b. Print the square of all numbers between 10 and 20 in the Transcript.
   c. Print all numbers that are between 1 and 1,000 and are divisible by 11 or 13 in the Transcript.
3. The factorial of a positive integer n is defined as follows: When n=1, the factorial of n is 1. When n>1, the factorial of n is n times the factorial of n-1. Write and test a recursive method derived directly from this definition. (The factorial method in the library is not defined recursively.)
4. The iteration methods covered in this section are all defined in class BlockClosure. Could they be redefined in the Boolean classes? If so, how?
5. What is the advantage of using a block rather than a Boolean expression as the receiver of a message?

4.10 Other forms of iteration

To close this chapter, we will introduce messages that strip the whileTrue and whileFalse ideas to a bare minimum by removing the argument. This means that all work done during the iteration must be done in the condition block. These messages have the following form:

aBlock whileTrue	"Repeats until aBlock evaluates to false."
aBlock whileFalse	"Repeats until aBlock evaluates to true."
aBlock repeat	"Repeats forever or until termination is forced inside aBlock."

When you examine the library, you will find that it does not use these messages. (The references that you will find are not message sends.) Yet, they are useful and most of the uses of whileTrue: and whileFalse: can be easily converted into this form.

Example 1: Find and print the largest integer number x for which $x + (270*x^2) < 100{,}000$

This problem can be solved either with

```
| x |
x := 1.
[x + (270 * x squared) < 100000] whileTrue: [x := x+1].
Transcript show: (x -1) printString
```

or with

```
| x |
x := 0.
[x := x + 1. x + (270 * x squared) < 100000] whileTrue.
Transcript show: (x - 1) printString
```

The first solution is more natural and easier to read which is probably why whileTrue is not used. To implement the same task with repeat, we would have to change the structure to

```
| x |
x := 0.
[x := x + 1.
        x + (270 * x squared) < 100000
        ifTrue: [Transcript show: (x - 1) printString. ^self]] repeat
```

where we force exit from the loop with the return operator.

Example 2: Finding the root of a function
Problem: We needs to find an approximate solution of

$1 + \sin(x) = x$

Solution: The problem can be solved by converting the equation into

$f(x) = 1 + \sin(x) - x = 0$

and finding the root of f(x) - the value of x that produces f(x) = 0. Since this approach can be used to find the solution of any kind of equation with one unknown, we decide to restate the problem as follows:

Problem: Define a general purpose method to solve

$f(x) = 0$

Solution: Perhaps the simplest approach is as follows (Figure 4.22): Start with a suitable first guess for x and calculate f(x). Increment x by a fixed increment and calculate f(x) again. If the new f(x) has the same sign as the old f(x), increment x. Keep repeating these steps until the new and the old value of f(x) have a different sign. At this point, decrease the size of the step (for example, divide it by 3), change its sign, and proceed as before (in the opposite direction, because the sign is different). Keep going back and forth until the value of f(x) becomes small enough or until the number of repetitions reaches a preset maximum.

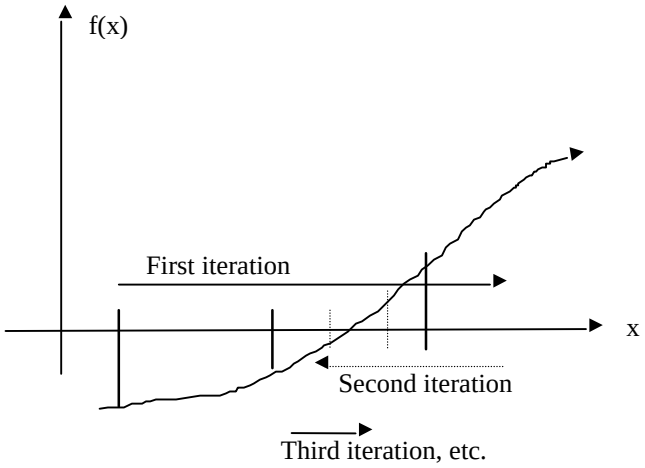

Figure 4.22. Finding root of f(x) = 0 by iteration.

We will now solve the problem in a simplified form by repeating the iteration until the two consecutive end-points are less than a predefined value delta apart. The algorithm is as follows:

1. Initialize starting value of x and use it as the current end-point.
2. Calculate the first value of f(x).
3. While the termination condition fails, repeat the following steps:
   a. While the old f(x) and the new f(x) have the same signs signs
      Increment x and calculate new f(x).
   b. Divide increment by 3 and change its sign.
   c. Re-evaluate the termination condition.
   d. Store x as the current end-point.

Before writing the method, we will first test this algorithm on the following code fragment:

```
| delta x oldx step oldF newF condition |
"Initialize."
delta := 0.01.
step := 0.1.
x := 0.
oldx := 0.
condition := false. "Termination condition - initially not satisfied."
newF := 1 + x sin - x.
[condition] whileFalse:
        [[x := x + step.                "Increment x and calculate new value."
        oldF := newF.
        newF := 1 + x sin - x.
        oldF sign = newF sign] whileTrue.
        step := (step / 3) negated.  "Update x and count."
        condition := (oldx - x) abs < delta.
        oldx := x].
"Output results."
Transcript clear; show: 'The approximate value of the root is x = ', x printString; cr;
        show: 'The value of the approximation is f(x) = ', newF printString
```

When we executed this code, we obtained the following output:

The approximate value of the root is x = 1.93457
The value of the approximation is f(x) = -6.67572e-6

This result is reasonable and we will thus convert the test fragment to a method.

The first question is what form the method should have. The method will need to know the function (this can be implemented as an evaluated block) and the following numbers: the starting point, the initial increment, and the value of delta. We thus decide on an implementation with selector rootFrom:by:untilDelta: that will be used as in

```
|function root |
function := [:x | 1 + x sin - x].
root :=  function rootFrom: 0.0 by: 0.1 untilDelta: 0.01.
Transcript clear;
        show: 'The approximate value of the root is x = ' , root printString;
        cr;
        show: 'The value of the approximation is f(x) = ' , (function value: root) printString
```

Note that we took the output out because the method could be used in situations in which no output is required.

The rest of this section is somewhat advanced for this point in the text and you may skip it without missing anything because we will return to the presented concepts later.

Where should the new method be defined? Our example shows that the argument is a block and so the definition will be in class BlockClosure. The implementation based on a few changes in our code fragment is as follows:

```
rootFrom: firstx by: firstStep untilDelta: delta
"Find root of self value = 0 starting from firstx in steps starting with firstStep. Use iteration back and forth
until x end points differ by less than delta."
        | x oldx step oldF newF condition |
        "Initialize."
        step := firstStep.
        x := firstx.
        oldx := 0.
        condition := false."Termination condition - initially not satisfied."
        newF := self value: x.
        [condition]
                whileFalse:
                        [
                        [x := x + step.       "Increment x and calculate new value."
                        oldF := newF.
                        newF := self value: x.
                        oldF sign = newF sign] whileTrue.
                        step := (step / 3) negated.  "Update x and count."
                        condition := (oldx - x) abs < delta.
                        oldx := x].
        ^x
```

This definition requires several comments:

- We could not use arguments firstx and firstStep in the iterations because arguments cannot be assigned new values inside a method. We thus had to introduce temporary variables x and step and initialize them to firstx and firstStep.
- The block that represents f(x) must have an argument so that we can evaluate f(x) for a particular value. To be able to supply an argument into a block, we must use a block with an internal argument as in

  ```
  [:x | 1 + x sin - x]
  ```

  Note the special syntax: x is inside the block, preceded by a colon and followed by a vertical bar. A block may have as many internal arguments as we need.
- To evaluate a block with an argument, we must send it an argument object and the value message is thus insufficient. For a block with one argument, use message value: instead as in

```
[:x | 1 + x sin - x] value: 0.4
```

When we tested the new definition with

```
|function root |
function := [:x | 1 + x sin - x].
root :=  function rootFrom: 0.0 by: 0.1 untilDelta: 0.01.
Transcript clear;
        show: 'The approximate value of the root is x = ' , root printString;
        cr;
        show: 'The value of the approximation is f(x) = ' , (function value: root) printString
```

we obtained the same result as before. The method is thus a correct translation of the original code fragment.

Main lessons learned:

- Methods whileTrue, whileFalse, and repeat don't have an argument block and all work must be done in the condition block..
- The use of repeat requires forced exit from the block.

Exercises

1. Solve the following problems from the previous section using whileTrue, whileFalse, and repeat:.
   a. Open a confirmer asking the user whether to stop execution or not. If the answer is yes, the program stops, otherwise the confirmer is displayed again.
   b. Print the square of all numbers between 10 and 20 to the Transcript.
   c. Print all numbers between 1 and 1,000 divisible by 11 or 13 to the Transcript.
2. Draw a flowchart describing the operation of the whileFalse method.

Conclusion

This chapter focused on Boolean objects and blocks. Booleans are used to represent the outcome of tests and yes/no questions. They are an essential component of most computer programs because they are the basis of decision making (should we execute a block of statements or not?), selection (should we execute block A or block B?), and iteration (should we execute the block again?).

In Smalltalk, Boolean objects are implemented by the abstract class Boolean which leaves the implementation of most of its methods to its subclasses True and False. Classes True and False each allow only one instance represented by literals true and false.

Another essential component of decisions, selections, and iterations is the block object, an instance of BlockClosure, a sequence of statements surrounded by square brackets. Class BlockClosure defines various forms of the value message which evaluate the statements inside the block in its proper context. Until a block object receives the value message, it is just a container holding a deferred sequence of statements.

Messages implementing decisions are defined in classes True and False. They include the one-keyword messages ifTrue: and ifFalse: and the two-key word messages ifTrue:ifFalse: and ifFalse: ifTrue:. All use blocks as arguments. Classes True and False also implement logic operations *and*, *or*, *exclusive or*, and *not*. Operations *and* and *or* are implemented in two forms - fully-evaluating and non-evaluating. The non-evaluating message does not evaluate its argument unless it is required to obtain the Boolean result. It is thus faster and preferable. Besides, it can be used as a safeguard protecting illegal evaluation of the argument.

Iteration methods are defined in class BlockClosure and their receiver block must evaluate to true or false. All forms of iteration defined in BlockClosure except for repeat require the evaluation of a condition block which determines whether to iterate again or proceed to the next message.

The body of the definition of whileTrue: re-sends the whileTrue: message, a technique known as recursion. Recursive solutions are often simpler than iterative ones but they are usually less efficient because of the extra work and memory overhead required by each new message send. In most cases, recursion can be easily replaced by more efficient iteration.

Although the Smalltalk library contains formal definitions of ifTrue:, ifFalse:, and whileTrue:, the compiler usually does not use them when compiling Smalltalk code. Instead, it uses pre-defined machine code that implements the operation more efficiently. This technique is called in-lining and its use in Smalltalk is rather rare. The vast majority of method definitions in the Smalltalk library are compiled into message sends reflecting directly the original code.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

BlockClosure, Boolean , *Date*, **Dialog**, **False**, *Time*, **True**.

Terms introduced in this chapter

and - Boolean operation; returns true if and only if the receiver and the argument are both true
Boolean - general term denoting true and false objects and operations on them; basis of tests
block - instance of class BlockClosure; sequence of statements surrounded by square brackets and evaluated together; to evaluate it, send it message value
body - last part of method definition - sequence of statements implementing method behavior
confirmer - a dialog window offering a yes/no choice
conjunction - another name for the *and* operation
conversion message - message converting an object into a related object
dialog - window-based interactive communication between the user and a program
disjunction - another name for the *or* operation
equality - two objects are equal if their values are in some sense interchangeable; the = message
equivalence - two objects are equivalent if they are identical, one and the same object; the == message
exclusive or - Boolean operation; returns true if and only if the receiver and the argument are different Booleans
file-in - reading and compiling a file-out file and inserting its contents into the library
file-out - storing a method, protocol, class, or category in a file using a special format suitable for a file-in
flowchart - a diagram showing the order of execution of an algorithm
fully-evaluating and/or - implementation of *and*/*or* in which both the receiver and the argument are always evaluated
heading - the first part of method definition ; specifies the selector and argument names, if any
in-line code – machine code inserted by the compiler into method translation instead of code to invoke a message; used for a few frequently used messages to increase speed of execution
iteration - repetition of a block of statements until some condition is satisfied
literal - an object created by the compiler from its textual form without a creation message
logic - synonym of Boolean
method definition - formal specification of the implementation of a message
non-evaluating and/or - implementation of *and*/*or* that evaluates the argument only if necessary
notifier - a dialog window displaying text and offering only an *OK* button for continuation; created with the warn: message to class Dialog
or - Boolean operation; returns false if and only if both the receiver and the argument are false
recursive – referring to itself

singleton - the single instance of a class that does not allow multiple instances
truth table - a table defining a Boolean operation by listing all combinations of *true* and *false* operand values and the corresponding value of the result