Chapter 19 - Smalltalk - an Object-Oriented Language

19.1 Overview

Smalltalk was created by the Software Concepts Group at the Xerox Palo Alto Research Center. It consists of a powerful user interface and environment in addition to its language features. In this book, we concentrate on the language itself, but the environment has had a significant impact on the computer world as well, serving as a model for modern window based systems.

The environment consists of windows, several of which can be on the screen at one time. Each window contains a different view of the present system. The user interacts with the Smalltalk environment through the use of pop-up menus that can be accessed through the use of a mouse. This environment greatly influenced the design of the Apple Macintosh, Microsoft Windows, and X Windows user interfaces as well as that of many other modern systems.

Smalltalk uses five standard system windows. Other types of windows can be created as classes by the user. The five standard windows are the System Browser, the System Workspace, the File List, the Workspace, and the System Transcript windows. We briefly describe the function of each of these windows below.

The System Browser presents a list of all classes in the system in one subview. The user may select any one of these classes for further examination. When a class is selected, a list of messages that apply to that class appears in another subview of the window. When one of these messages is selected by the user, the method associated with the message is displayed in yet another subview. Through this window, the user may add or delete classes, add or delete messages, and modify methods.

The System Workspace window contains Smalltalk definitions of various utilities. Through this window the user can examine and modify these definitions.

The File List window presents a subview consisting of a directory from secondary storage. Entries in the directory may be selected, in which case the selected file appears in a second subview. The text of the file can then be edited in that subview.

The Workspace window can hold temporary text. It is used as a scratchpad area.

The System Transcript window contains all messages and output from the Smalltalk system.

Smalltalk language implements all of the fundamental object-oriented concepts. In particular, every entity in Smalltalk is an object and all operations are activated by sending messages to objects. The concept of class is fundamental. Every object is identified as having a defining class. In addition, inheritance is implemented through the definition of subclasses and super classes.

Smalltalk has a very simple syntax, consisting of only one basic type of statement: message sending. Collections of message sending statements are used for the sole purpose of defining methods for classes. Classes are defined through the interactive environment by binding a name to the class, variable names to the class, messages to the class, and a method to each message. The syntax of Smalltalk is introduced later.

The power of Smalltalk comes from its built-in hierarchical collection of classes, called the Smalltalk image. These classes, though an integral part of the language, are treated like all other classes in that, being expressed in Smalltalk themselves, they may be deleted or modified by the user through the interactive Smalltalk environment. The user also has the ability to add newly-created classes to the image. The structure of the Smalltalk image is described at a later point.

19.2 Smalltalk Syntax

Every statement in Smalltalk is a message sending statement and appears within the definition of a method. The general form of a statement is

<statement> ::= <object> <message>

A period is used as a statement separator in Smalltalk. We next examine the form of each of the two component parts
19.2.1 Objects

Objects can be expressed in four different forms: literals, reserved words, variables, and expressions.

19.2.1.1 Literals

Smalltalk provides literals for representing instances of five different classes. Character literals are represented by a dollar sign followed by any ASCII character. Examples are

```
$a  $?  $$  $*  $Z
```

Numeric literals are written with digits, a minus sign, a decimal point, a lower case letter e for expressing a power of ten, and a slash to represent fractions. The class of a numeric literal is determined by its format. For example, the following literals represent objects of class **Integer**:

```
1  -14  217
```

The following numeric literals represent objects of class **Float**:

```
6.28  -14.2  .7e12  33.5e-10
```

The following numeric literals represent objects of class **Fraction**:

```
1/7  -7/721  4/9
```

Literals representing objects of class **String** are strings of characters enclosed in single quotes. For example,

```
'This is a String literal'
'Special @+:$ characters are allowed'
'It''s also possible to include single quotes'
```

Literals of class **Symbol** are sequences of characters following a number sign (#). Each symbol must be unique. For example:

```
#aSymbol  #=  #$S
```

An **Array** literal is a sequence of objects. The objects need not all be of the same class and they need not be literals. They are written by enclosing the sequence in parentheses, separating individual elements by spaces, and preceding the left parenthesis with a number sign (#). Examples of array literals are:

```
#(1 $a v)  #(example' 2.4 7/9 $*)
```

19.2.1.2 Reserved Words

In addition to literal representation of objects, there are five reserved words that represent specific objects in
Smalltalk.

First, true and false represent the only objects of classes True and False respectively. True and False are subclasses of class Boolean.

The reserved word nil represents the only instance object of the class UndefinedObject. Variables are bound to this object when they are created indicating an undefined state.

The reserved word self, when used in the definition of a method, is a pseudovariable that always refers to the receiving object of the message. The reserved word super also refers to the receiving object of the message, but with the object considered as an instance of the super class of its defining class.

19.2.1.3 Variables

Within Smalltalk, there are six kinds of variables. These are all represented by identifiers consisting of a sequence of letters. No special characters are permitted in Smalltalk variable names. The convention used to express multiword names is to capitalize the first letter of every word after the first word. The first letter of an instance variable, temporary variable, or parameter is, by convention, always lowercase. The first letter of a class variable, class name, or global variable is always uppercase.

Instance variables are defined with a class definition and are private to each instance of that class. Each instance of a class, therefore, has the same set of instance variables, though the objects associated with these variables may differ from instance to instance. An instance object's instance variables are accessible only from the instance methods for that object.

Class variables are defined within a class and are accessible to every instance object of that class and its descendant classes as well as from the class methods of the class and all of its descendant classes.

Temporary variables are defined only during the execution of a single method in which they are defined. When the method is completed, these variables cease to exist. These variables are declared at the beginning of a method definition by listing their names enclosed between a pair of vertical bars.

Global variables are accessible to all methods in the Smalltalk system. These names are listed in the system dictionary.

Pool variables are shared among classes that specify them. They are listed in named pool directories.

Parameters represent objects passed into a method when it is selected by its corresponding message. They are private to the method and redefined upon every method activation.

Variables are initially bound to the nil object, the only instance of class UndefinedObject. They are then bound to other objects by the assignment message. The class of a variable is the class of the object that was most recently assigned to it. Therefore variables are dynamically bound to their classes.

The assignment of an object to a variable is specified in Smalltalk by using the left-arrow as the assignment message. An object is specified to the right of the assignment message, and that object is bound to the variable on the left as a result of the assignment message. The variable is bound to both the class and the value of the object found on the right side of the arrow.

For example,

\[ a \leftarrow 6 \]

would bind the variable \( a \) to class Integer and object 6. Variable bindings are indirect in the sense that a pointer to the object is associated with the variable. For example,

\[ a \leftarrow b \]

binds the variable \( a \) to the same class and object which is currently bound to variable \( b \). It does not create a copy of the object.
19.2.1.4 Expressions

When a message is sent to an object in Smalltalk, the corresponding method returns an object as a result of its handling the message. In this way, the method behaves in a manner similar to a function. The returned object can then act as the recipient of a second message. The grouping of several messages into a single statement results in the construction of Smalltalk expressions.

For our discussion of expressions, we work with objects of class Integer and the messages +, , and - which are appropriate to that class. Note that operators are taken to be special forms of messages. Therefore, the expression

$$6 + 4$$

means that the message + is sent to 6, a literal object of Integer class with parameter 4. The return value of this message is the sum 10, another Integer object.

Therefore, the expression $$6 + 4$$ represents an integer object returned from the handling of this message. Suppose we wish to multiply the result of the above message by 2. We can do so with the following nested message:

$$6 + 4 * 2$$

This causes the message * to be sent to the object 6+4 with parameter 2. The resulting Integer object would be 20.

Note that this is in opposition to the conventional precedence rules for arithmetic. The message-sending protocol of Smalltalk results in a strict left-to-right evaluation. Smalltalk does, however, permit the grouping of messages through parentheses. For example, the message

$$6 + (4 * 2)$$

returns the Integer object 14.

19.2.2 Messages

Smalltalk permits three types of messages. They are unary, binary, and keyword and are distinguished by the number of parameters and the way the message is specified.

19.2.2.1 Unary Messages

Unary messages are parameterless. Such messages are specified by identifiers with the first letter uppercase for a class message and lower case for an instance message. Examples are

$$6 \text{ factorial}$$
$$12 \text{ negated}$$

19.2.2.2 Binary Messages

Binary messages are named by one or two special characters and always require a single parameter which follows the symbol which represents the message. Examples of binary messages are

$$6 + 5$$
$$'one', 'two'$$
19.2.2.3 Keyword Messages

Keyword messages consist of one or more keyword/formal parameter pairs. Both the keywords and the parameters are identifiers, and a keyword is separated from its corresponding formal parameter by a colon. The keywords for methods are chosen to make the reading of the message as explanatory of its function as possible. The message name is always given as the sequence of keywords, each followed by a colon. The number of parameters accepted by a message is always equal to the number of colons in its name. The mode of parameters is \textit{IN} implemented by copy though one should keep in mind that the parameters are bound to pointers to objects. Examples of keyword message calls are

\begin{verbatim}
6 gcd: 21
'test string' at:3 put:$x
\end{verbatim}

19.2.3 Blocks

One type of Smalltalk object that has not yet been described corresponds to the block mentioned for HOOL. A Smalltalk block is a sequence of message specifications that is executable. A block literal is specified by placing the sequence of messages, separated by periods, inside of square brackets. Blocks are important tools for specifying conditional and iterative messages. But before we look at that use, let's illustrate the use of blocks for deferred execution.

Suppose we define the block

\begin{verbatim}
zeroOut ← [a ← 0. b ← 0. c ← 0]
\end{verbatim}

This binds the variable \texttt{zeroOut} to the block of messages that bind \texttt{a}, \texttt{b}, and \texttt{c} all to the \texttt{Integer} object \texttt{0}. This variable of class \texttt{Block} can then be executed at any time by sending the message \texttt{value} to \texttt{zeroOut} by

\begin{verbatim}
zeroOut value
\end{verbatim}

The key concept here is that blocks of messages can themselves be considered to be objects. One important message that can be sent to such an object is \texttt{value}, illustrated above, which requests the block to execute itself.

This leads us to examine the Smalltalk approach to conditional execution. It consists of sending the message \texttt{ifTrue:} to a Boolean object (either \texttt{true} or \texttt{false}) and executing the single parameter of the message, a block object, if the receiver is true.

For example, the message

\begin{verbatim}
(x > y) ifTrue: [x ← y]
\end{verbatim}

results in \texttt{x} being assigned the object bound to \texttt{y} if the value presently bound to \texttt{x} is greater than the value presently bound to \texttt{y}.

Similarly, the \texttt{ifFalse:} message executes the parameter block if the receiving object is false. Often the two messages \texttt{ifTrue:} and \texttt{ifFalse:} are sent consecutively to the same object. When consecutive messages are sent to the same object, the messages may appear after the receiving object separated by semicolons. For example, the following message sequence sets \texttt{z} to the maximum of \texttt{x} and \texttt{y}:

\begin{verbatim}
(x > y) ifTrue: [z ← x];
    ifFalse: [z ← y]
\end{verbatim}
Chapter 19 - July 26, 2002 Draft

This is equivalent to

\[
(x > y) \text{ ifTrue: } [z \leftarrow x]. \\
(x > y) \text{ ifFalse: } [z \leftarrow y]
\]

Alternatively, Boolean objects can receive the single message \texttt{ifTrue:ifFalse:} which contains two parameters of class \texttt{Block}. The above result can then be obtained by

\[
(x > y) \text{ ifTrue: } [z \leftarrow x] \\
\text{ ifFalse: } [z \leftarrow y]
\]

Repeated execution of a block object is specified by the \texttt{whileTrue:} and \texttt{whileFalse:} messages. The receiving object of these messages if a block that returns a Boolean object when executed. The general form of the \texttt{whileTrue:} message is

\[
<\text{receiving-block}> \text{ whileTrue: } <\text{parameter-block}>
\]

The resulting action is the repeated evaluation of the \texttt{receiving-block} followed by the execution of the \texttt{parameter-block}, until the \texttt{receiving-block} no longer results in a true object. The following message sequences use the \texttt{whileTrue:} message to set \texttt{logn} to the largest integer less than or equal to \(\log_2 n\).

\[
\text{logn} \leftarrow 0. \\
\text{temp} \leftarrow n \text{ div } 2. \\
[\text{temp} > 0] \text{ whileTrue: } [\text{temp} \leftarrow \text{temp} \text{ div } 2. \\
\text{logn} \leftarrow \text{logn} + 1]
\]

Smalltalk also contains messages that implement variable controlled iterations, but we do not discuss those here.

Figure 19.1 Smalltalk Method Description

1     longident 
2     "This method computes the longest substring of 
3         identical characters within the receiving string" 
4     |n long lastchar consec| 
5     n \leftarrow 1. 
6     long \leftarrow 0. 
7     lastchar \leftarrow self at: 1. 
8     consec \leftarrow 0. 
9     [n <= (self size)] whileTrue: 
10    [(lastchar = (self at: n)) 
11    \text{ ifTrue:} [consec \leftarrow consec + 1] 
12    \text{ ifFalse:} [lastchar \leftarrow self at: n. 
13    (consec > long) 
14    \text{ ifTrue:} [long \leftarrow consec]. 
15    consec \leftarrow 1]. 
16    n \leftarrow n + 1]. 
17    \uparrow \text{long}
19.2.4 Definition of a Method

A method is defined as a sequence of messages separated by periods. Two special features of Smalltalk method definition should be observed. First, a method may contain variables which are local to the execution of the method. These are declared on the first line of the message with the variable names separated by spaces and the entire list enclosed in vertical bars. As mentioned previously, no class needs to be specified in this declaration since classes of variables are determined dynamically as they are assigned objects.

The second feature is the ability to specify a return value for the method. This is done by preceding an object description by an up-arrow. The object following the up-arrow is then returned as the result of the message.

The method description is Figure 19.1 provides an example of the preceding features. The lines are numbered to facilitate reference in the following discussion and are not a part of the Smalltalk syntax.

In line 4, four local variables are declared. The location bindings for these variables hold for the duration of the method execution. Like all Smalltalk variables, they are bound to a class whenever they are assigned an object. Prior to that they are bound to object nil of class UndefinedObject.

The variable \( n \) is an index into the string, recording the position of the current character of the string scan. The variable \( long \) is the length of the longest string of identical characters found. The variable \( lastchar \) remembers the last character scanned, while \( consec \) keeps a count of the number of consecutive identical characters at any point.

Line 16 indicates that the object bound to variable \( long \) is returned. Any object description preceded by the up-arrow indicates not only that the following object is returned, but also that the method is immediately terminated after the corresponding message returns its value.

19.2.5 Definition of a Class

Class descriptions in Smalltalk consist of the following components:
1. **Class name**: the name of the class, which by convention is chosen to begin with an uppercase letter.
2. **Super class**: The name of the immediate super class of the class being defined. The new class inherits variables, messages, and methods from this super class.
3. **Class variables**: These are variables that belong to the class and are accessible to all instances of this class.
4. **Instance variables**: Each instance object belonging to the defined class has a set of these instance variables that are accessible only to the methods of the owning instance.
5. **Class methods**: Methods that are activated by messages to the class object.
6. **Instance methods**: Methods that are activated by messages to instances of the class.

We illustrate this protocol description of a class by defining the class \( \text{Amount} \), a class to represent an amount of money, consisting of dollars and cents. The definition of this class is found in Figure 19.2. Line 1 indicates that \( \text{Amount} \) is a subclass of class \( \text{Object} \), and hence inherits the protocol of \( \text{Object} \), a super class of all Smalltalk classes.

\( \text{Class Amount} \) has no class variables and one instance variable \( \text{totalCents} \), declared in line 2. This variable is bound to an object of class \( \text{Integer} \) which is the total amount of cents associated with the amount of money.

\( \text{Amount} \) has one class method, \( \text{Dollars:Cents:} \) which is used to instantiate an instance of this class. It is defined in lines 7-8. Line 7 is the message prototype, specifying \( d \) and \( c \) as formal parameters of the method. In line 8, \( \text{self} \) refers to the class object \( \text{Amount} \) which is the only permissible recipient of this message. When \( \text{self} \) is sent the message \( \text{new} \), this creates a new instance of class \( \text{Amount} \). \( \text{Class Amount} \) inherits the class method \( \text{new} \) from its super class \( \text{Object} \). This newly created object is then sent the message \( \text{dollars: d cents: c} \) to establish its initial value. This is a permissible message to objects of class \( \text{Amount} \) and is defined in lines 38-39.

Nine instance methods are defined for class \( \text{Amount} \), four callable by binary messages, three by unary, and two by keyword.
Figure 19.2 Smalltalk Definition of Class Amount

1     Object subclass: #Amount
2       instanceVariableNames: 'totalCents'
3       classVariableNames: ''
4
5     !Amount class methods !
6
7     Dollars: d Cents: c
8       self new dollars: d cents: c
9
10    !Amount instance methods !
11    * aValue
12       | newTotalCents |
13       newTotalCents ← totalCents * aValue.
14 ↑    Amount Dollars: (newTotalCents rem: 100)
15        Cents: (newTotalCents quo: 100)
16
17    + anAmount
18       | newTotalCents |
19       newTotalCents ← totalCents + anAmount totalCents.
20 ↑    Amount Dollars: (newTotalCents rem: 100)
21        Cents: (newTotalCents quo: 100).
22
23    - anAmount
24       | newTotalCents |
25       newTotalCents ← totalCents - anAmount totalCents.
26 ↑    Amount Dollars: (newTotalCents rem: 100)
27        Cents: (newTotalCents quo: 100).
28
29    > anAmount
30      ↑ totalCents > (anAmount totalCents)
31
32    cents
33      ↑ totalCents rem: 100
34
35    dollars
36      ↑ totalCents quo: 100
37
38    dollars: d cents: c
39       totalCents ← 100 * d + c
40
41    printOn: aPrintStream
42       self dollars printOn: aPrintStream.
43       $. printOn: aPrintStream.
44      (self cents < 10) ifTrue: [$0 printOn: aPrintStream].
45      self cents printOn: aPrintStream
46
47    totalCents
48      ↑ totalCents
The binary message * permits the multiplication of an amount by any object of a numeric class. Note that
the parameter aValue could be of class Integer, Float, or Fraction, or any other class for which multiplication
by an integer is defined. The variable newTotalCents is declared to be local to method * on line 12. This variable
is bound to the result of multiplying the instance variable totalCents by the formal parameter aValue in line 13.
Lines 14 and 15 create a new instance of class Amount with its value represented by newTotalCents.

Similarly, the methods for binary messages + and - are defined in lines 17-21 and 23-27, respectively. The
binary message > is used to compare two amounts and returns the Boolean result of sending the > message to
totalCents, an instance of class Integer.

The methods for the three unary messages--cents, dollars, and totalCents--return the cents part of
the amount, the dollar part of the amount, and the total number of cents in the amount.

Two methods are activated by keyword messages. The first, dollars:cents:, is defined in lines 38-39.
It accepts two parameters and modifies the totalCents variable of the receiving object accordingly. The second,
printOn:, accepts the name of a print stream as its single parameter and sends the appropriate characters for the
amount to that print stream. Line 44 is required to support the printing of a leading zero character if the number of
cents is less than 10.

19.2.6 Creation of Instance Objects

Each class in Smalltalk is itself considered to be an object which receives messages. This class object can
receive any of the class messages specified for the class. The most important of the messages that the class object
receives is the new message, which creates a new instance of the specified class.

For example, an instance of class Amount defined in Figure 19.2 can be created and bound to a variable a
by the message

a ← Amount new

In this sense, the class object can be thought of as a factory for instance objects. On receipt of the new message, the
class object manufactures an instance object of the defining class, returning it as the result of the message.

It is also possible to create additional methods that not only manufacture new instance objects but also ini-
tialize those objects. The class method Dollars:Cents: defined in lines 7-8 of Figure 19.2 is an example of this.

19.3 Class Hierarchy

The image of a Smalltalk system is a hierarchy of classes consisting of a tree representing the subclass/super
class structure used for inheritance. Many of the classes in Smalltalk are never intended to have instance variables.
Rather, these abstract classes provide variables and methods that are to be inherited by a set of descendant classes.
Each Smalltalk system comes with a large class hierarchy already built into the image. The user can then further
enhance the image by adding new classes to this structure and modifying existing classes. These concepts are illus-
trated in the following sections.

19.4 Abstract Classes

We illustrate the concept of abstract class through the introduction of an example. Our example is the class
Number which is part of Smalltalk's built-in image.

Number serves as a super class for numeric classes like Integer, Float, and Fraction. As such, its
variables, messages, and methods are all inherited by these subclasses, so it defines those entities that are common to
all of its subclasses. Because Number is an abstract class, there are never any instances of this particular class. It only
exists to provide a template for its subclasses and their descendants.

In order to illustrate the different ways inherited methods are used, we examine five instance methods of
class Number and the manner in which they are inherited by subclass Float.

First we consider the method for the binary message +. It is defined in abstract class Number, but its definition there indicates that the actual definition is deferred to subclasses. The message + needs to exist as a valid message in class Number because other methods of class Number use that message. However, since Number is an abstract class and has no instances, each instance that receives the message + is an instance of some subclass or further descendant of Number such as Float where + does have a complete definition.

Next, the unary message abs, which defines the absolute value operation is defined for the abstract class Number and inherited by its subclasses as is, without any redefinition. For example, abs could be defined as

```smalltalk
abs
    self < 0 ifTrue: [↑self]
    ifFalse: [↑self negated]
```

The messages < and negated are messages that, like + discussed above, are defined in Number but whose implementation is deferred to subclasses.

The message numerator, which is defined for class Number, is also inherited by class Float. But this is an example of a message that although inherited, has no meaning for an instance of that subclass since numerator does not make sense for a Float instance.

The message > is an example of a message that is inherited from Number by subclass Float in a way that is transparent in the definition of Number. By this we mean > is defined in the class Magnitude which is the superclass of Number. Number then inherits > from Magnitude and Float inherits it from Number.

Finally, Float contains instance messages that are unique to that class and not inherited at all. For example, the method associated with message truncate returns the truncated integer part of a floating point number. This method is defined only in Float.

We summarize the five different ways message inheritance can occur in the following table:

<table>
<thead>
<tr>
<th>Type of inheritance</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined in super class but not implemented there</td>
<td>+</td>
</tr>
<tr>
<td>Defined in super class and inherited by subclass w/o redefinition</td>
<td>abs</td>
</tr>
<tr>
<td>Defined in super class and inherited by subclass but meaningless in subclass</td>
<td>numerator</td>
</tr>
<tr>
<td>Defined in super class of super class and inherited by subclass through its super class</td>
<td>&gt;</td>
</tr>
<tr>
<td>Defined in subclass only</td>
<td>truncate</td>
</tr>
</tbody>
</table>

### 19.4.1 Important Smalltalk Abstract Classes

Each version of Smalltalk comes with a different set of abstract classes provided in its initial image. But there are certain abstract classes that have become more or less standard components of all Smalltalk implementations, serving as the building blocks for all other classes. We briefly summarize some of these classes in this section.

Magnitude abstract class provides the protocol for subclasses that are linearly ordered. Its immediate subclasses include Date, Time, Character, and Number. The methods of class Magnitude that are inherited by all of its subclasses include ordered comparisons and determining the larger and the smaller of two objects.

Stream is an abstract class for subclasses of positionable collections. These are used for accessing files or other input/output devices in a sequential manner. The messages of Stream allow accessing or updating the object at the current position and messages to change the current position in the stream. Subclasses of Stream include
ReadStream, WriteStream, ReadWriteStream, FileStream, PrintStream, and TerminalStream.

Collection is an abstract class that represents either a structured or unstructured collection of objects. Many different types of collections with widely varying organizations make up the subclasses of Collection. Some of the more popular ones are Array, String, Set, and Dictionary. Different subclasses permit their elements to be accessed in different ways such as by position or by key.

Smalltalk also commonly contains abstract classes that support the use of window and graphic objects.
Figure 19.3 Class description of Account in Smalltalk

Object subclass: #Account
  instanceVariableNames:
    'balance monthlyReport name acctID'
  classVariableNames:
    'TotalBalance NumberOfAccts '
!Account class methods!  initialize
  TotalBalance ← Amount new.
  TotalBalance dollars: 0 cents: 0.
  NumberOfAccts ← 0

reportTotBal
  |totReport|
  totReport ← WriteStream with: String.
  totReport cr.
  'Number of Accounts = ' printOn: totReport.
  NumberOfAccts printOn: totReport.
  ↑totReport contents
!Account methods!

balance
  ↑balance

deposit: itemID of: anAmount on: date
  TotalBalance ← TotalBalance + anAmount.
  balance ← balance + anAmount.
  monthlyReport nextPutAll: 'DEP';
    tab;
    nextPutAll: itemID;
    tab.
  anAmount printOn: monthlyReport.
  monthlyReport tab;
    nextPutAll: date;
    tab.
  monthlyReport cr.

monthend
  |temp|
  temp ← monthlyReport contents.
  monthlyReport ← WriteStream with: String.
  ↑ temp

open: ID for: custName
  NumberOfAccts ← NumberOfAccts + 1.
  balance ← Amount new.
  balance dollars: 0 cents: 0.
  monthlyReport ← WriteStream with: String new.
name ← custName.
acctID ← iD.
monthlyReport nextPutAll: acctID; cr

withdraw: itemID of: anAmount on: date
(anAmount > balance)
   ifFalse: [balance ← balance - anAmount.
   TotalBalance ← TotalBalance - anAmount.
   monthlyReport nextPutAll: 'WTH'
    tab;
   nextPutAll: itemID;
    tab.
   anAmount printOn: monthlyReport.
   monthlyReport tab;
   nextPutAll: date;
    tab.
   monthlyReport cr.]
   ifTrue: [↑'Account overdrawn - withdrawal rejected']

19.5 An Example in Smalltalk

Figure 19.3 lists the Smalltalk version of the class Account. This is a direct translation of the HOOL version of the same class which was listed in Figure 14.1.

This class has four instance variables, two class variables, two class methods, and four instance methods. We examine these in some detail and then we examine subclasses Checking and Savings of class Account.

19.5.1 Class Account

Class Account is an abstract class. Two class methods initialize and reportTotBal are defined in Figure 19.3. The class method for initialize sets the class variable TotalBalance to an object of class Amount (as defined in Figure 14.4) and initializes that object to zero. This represents the sum of the balances of all instance accounts. This class method also sets class variable NumberOfAccts to an object of class Integer whose value is zero. This represents the total number of account instances that have been opened. The method associated with class message ReportTotBal constructs a report as an instance of class WriteStream and reports the total balance and the total number of accounts.

The instance method for deposit:of:on: is a method with three parameters. The first is an identification string for the deposit, the second the amount of the deposit, and the third the date of the deposit.

Message withdraw:of:on: sends as parameters the identification string, the amount of withdrawal, and the date of withdrawal. It also contains a test for the amount of withdrawal greater than the account balance to prevent overdrawing the account.

The method associated with open:for: creates a new instance of class Account and initializes its balance, monthly report, account ID, and customer name.

The method associated with instance message monthend sends the report accumulated in monthlyReport to the printer and clears monthlyReport for the next month.

19.5.2 Class Checking
Figure 19.4 Class description of Checking in Smalltalk

In Figure 19.4, class Checking is defined as a subclass of class Account. In addition to the protocol inherited from Account, Checking has one additional class variable, one class method, and two instance methods.

The class variable added is MonthlyServiceCharge which is given its value in the new class method SetServiceCharge.

The added instance method is check:of:on and is simply a renaming of the instance method withdraw:of:on from class Account.

The instance method monthend: inserts the withdrawal of the service charge before sending the message monthend whose method is inherited from class Account. Note that Smalltalk distinguishes between the two messages monthend and monthend:.

19.5.3 Class Savings

Class Savings is defined in Figure 19.5. It adds the class variable InterestRate which is set by class method SetInterestRate.

Instance method monthend: includes sending the message calculateInterest which for simplicity, calculates the interest on the monthly closing balance.
Figure 19.5 Class description of Savings in Smalltalk

Account subclass: #Savings
  instanceVariableNames: ''
  classVariableNames: 'InterestRate'

!Savings class methods!
setInterestRate: aFloat
  InterestRate ← aFloat

!Savings instance methods!
calculateInterest
  |totalCentsInt interest|
  totalCentsInt ← ((InterestRate / 1200) * (balance totalCents)) truncated.
  interest ← Amount new.
  interest dollars: (totalCentsInt / 100)
  cents: (totalCentsInt rem: 100).
↑interest

monthend: date
  |interestAmt|
  interestAmt ← self calculateInterest.
  self deposit: 'INT' of: interestAmt on: date.
↑self monthend.

Chapter 19 - Terms
unary messages
binary messages
keyword messages
instance
Smalltalk image
subclass
super class
abstract class

Chapter 19 - Discussion Questions
1. Smalltalk permits objects to be returned as a result of sending a message. How does this increase the power of these languages?

2. Contrast Smalltalk's manner of specifying multiple parameters (via keywords and colons) to that found in HOOL and imperative languages. Which do you prefer and why?

3. What is the difference between the relationship of a class to its superclass and the relationship of an object to its defining class?

4. How is an unconstrained array in Ada similar to an abstract class in Smalltalk? How is a generic package similar to an abstract class?

5. What are some advantages and disadvantages of having a modifiable built-in image in Smalltalk?

6. How is the concept of a block in Smalltalk different from the concept of block in an imperative language? How are
they the same?

Chapter 19 - Exercises

1. The Smalltalk class Fraction has two instance variables, denominator and numerator, both of class Integer. Write the class method numerator: denominator which defines the value of a Fraction object.

2. Write the following instance methods for the class Fraction defined in Exercise 1:
   (a) * aNumber
   (b) + aNumber
   (c) - aNumber
   (d) / aNumber
   (e) < aNumber
   (f) asFloat
   (g) denominator
   (h) numerator
   (i) reciprocal

3. What will be the result of each of the final Smalltalk message in each of the following:
   (a) a ← 3. a + 4
   (b) a ← 3. a + 4; -2
   (c) a ← 3. a / 2 + 1
   (d) a ← 3. a / (2 + 1)
   (e) a ← 3. a + 6 / 2

4. If each of the following is preceded by
   a ← 5.
   b ← 2.
what will be returned in each of the following cases?
   (a) [a < b] ifTrue: [a + 1]
       ifFalse: [b - 1]

   (b) [b < a] whileTrue: [a ← a - 1. b * a]

   (c) x ← 0.
       [n > 1] whileTrue: [x ← x + n. n ← n - 1. x]

5. Write a Smalltalk method for each of the following on class Integer:
   (a) sumOfFirstN
   (b) factorial
   (c) greatestCommonDivisor:
   (d) max:
   (e) floorOfSquareRoot

Chapter 19 - Laboratory Exercises

1. Write a class definition for a stack in Smalltalk.
2. Write a class definition for a queue in Smalltalk.

3. Write a class definition for a binary search tree in Smalltalk.

4. Write a class definition for real open intervals in Smalltalk. Objects of class Interval should accept the following messages:

   left       return the left end point
   right      return the right end point
   in: aFloat return true if aFloat is in the interval
   in: anInterval
            return true if anInterval is contained
            in the receiving interval