



Johannes Brauer

Programming Smalltalk – Object-Orientation from the Beginning

An introduction to the principles
of programming



Springer Vieweg

Deliver Perfect Applications Faster

Cincom Smalltalk™

- Pure object orientation – “everything is an object”
- Designed for a dynamic world – dynamically typed, dynamic language, dynamic system
- Modern platform for any kind of application – rich client, server, Web, SOA, mobile
- Based on open industry standards
- Instantly binary portable across Windows, OS-X, Linux and UNIX
- Ideally suited for agile processes
- Business-critical applications with millions of users worldwide and across all sectors including banking, insurance, automotive and electronics – many have been operating for more than 15 years



What are your benefits?

- Easy to learn
- Simple to use
- Less errors
- More enjoyable to use
- Become part of an exciting community. Those who have programmed in Smalltalk once never want to do without it!

Happy Smalltalking!

Further information: www.cincomsmalltalk.com/VisualWorks

Please share your Smalltalk experiences with us:

Cincom Systems, Inc.

Tel: 1-800-2CINCOM

Tel: 1-513-612-2300

Email: info@cincom.com

<http://www.cincom.com>



Johannes Brauer

Programming Smalltalk – Object-Orientation from the Beginning

An introduction to the principles
of programming



Springer Vieweg

Johannes Brauer
NORDAKADEMIE, Hochschule der Wirtschaft
Elmshorn, Germany

1st English edition (based on the 4th revised and expanded German original)
The translation and production of this book was sponsored by Cincom Systems.

ISBN 978-3-658-06822-6
DOI 10.1007/978-3-658-06823-3

ISBN 978-3-658-06823-3 (eBook)

Library of Congress Control Number: 2015931398

Springer Vieweg
© Springer Fachmedien Wiesbaden 2015
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer Fachmedien Wiesbaden GmbH is part of Springer Science+Business Media (www.springer.com)

Foreword

Smalltalk is an outstanding technology. It has been in the forefront of many innovations that have helped to advance the art of application software development such as GUIs with overlapping windows, the mouse, handheld devices, web applications, object-oriented programming, test-driven development and extreme programming, just to name a few. Smalltalk was at the core of a revolution to enable true personal computing. This revolution would eventually lead to the creation of a generic device that would allow people to help solve their problems back in the late 1960s and 1970s, an era that knew only large centralised data centres.

During this same timeframe, in 1968, Cincom Systems was founded with then revolutionary ideas of simplifying the complexity associated with developing business. We saw that Smalltalk was the next evolution in application development, and that we share the same philosophy with Smalltalk: to bring value to users by providing simple, yet powerful solutions. Consequently, we bought into this technology in the mid-1990s and have continued to invest in it ever since.

Advanced application technology is nothing without advanced users. Cincom is convinced that good education is the foundation for a better future. To support this on a worldwide basis, we established the Cincom Smalltalk Academic Program that allows academic institutions to use Cincom Smalltalk as part of a teaching curriculum free of charge. One such institution is the German university “NORDAKADEMIE” where Professor Brauer teaches his students—in a practice-oriented way—how to write software. Professor Brauer had developed his own introductory course to object-oriented programming using Cincom Smalltalk, and he made the course publicly available through his German textbook “Grundkurs Smalltalk—Objektorientierung von Anfang an”.

It was an honour for us to support Professor Brauer with the translation of his book to English. Now his excellent course is available to students and teachers all over the world, nicely complementing the Cincom Smalltalk Academic Program.

We wish all readers a lot of fun and success with Smalltalk!



Brian L. Bish, Managing Director, Cincom Smalltalk

Preface

The origins of this book lie in supplementary material for a basic programming course that is offered in the first two semesters of the commercial IT course of study at the NOR-DAKADEMIE Business School. Since the winter semester of 1999/2000, *object-oriented programming* has been taught using the *Smalltalk* programming language. It was always difficult to decide which text books to assign to students, because the many books that were available to teach object-oriented programming in general and Smalltalk in particular often presupposed at the very least programming knowledge in a non-object-oriented programming language. Even though the number of beginning students who have such knowledge continues to increase, it's not something that one can generally assume to be true. In fact, when the first edition of this book appeared, texts for beginning programming students generally did not teach object-oriented programming. In the meantime, even though the “Objects-First” approach has become increasingly common, it's still relatively rare to encounter Smalltalk as a first programming language. For that reason, this book continues to fill a gap. It is intended not just for IT students, but for anyone who seeks a fundamental entry point for programming, especially object-oriented programming. For example, it can also be used for advanced IT courses in the later grades of secondary schools.

Choosing Smalltalk is important above all for pedagogical reasons. Smalltalk is a simple, strictly object-oriented language that practically forces one to think in an object-oriented way. In addition, almost all students can be said to suffer the same disadvantage, that of not knowing this language, which somewhat reduces the problems caused by the different levels of initial knowledge among the students. And furthermore, a variety of free development environments are available, which students can easily install on their own computers.

With regard to its use in the real world of industry, Smalltalk is certainly not as important as, say, Java. On the other hand, so-called *dynamic* languages like Python and Ruby, which adapted many of their ideas from Smalltalk, are enjoying increasing popularity and wider distribution. Smalltalk is becoming increasingly important as an introductory programming language for children in elementary schools. In this context, one can mention

the *Etoys* and *Scratch* projects that have both been implemented in the Smalltalk dialect Squeak. Squeak runs on nearly all available operating systems as well as on the OLPC¹ XO computers (“\$100 laptops”).

This book, however, is not a programming-language course in a narrow sense. Above all, it should not be treated as a complete presentation of the extensive Smalltalk class library. For that, one must consult the relevant documentation for the development environment being used. Nevertheless, it’s also necessary to treat the basic concepts of a development environment, since Smalltalk programming always occurs within one. For beginners in particular, this can be an additional impediment. Stated somewhat simply, it used to be enough to be able to use a text editor and a compiler; now, besides learning the basics of programming, a student must also learn technical skills for dealing with a complex development environment.

An introduction to programming must also include fundamental knowledge of how to construct algorithms—a theme that is usually not dealt with in connection with object orientation. This is necessary because it is only through concerning oneself with the elementary problems of programming that one can develop an understanding for how computers work.

The first chapter teaches readers enough basic concepts of computer science so that they can begin to learn programming. Then, in the second chapter, a simple example provides an initial introduction to the basic construction of algorithms and how to construct them in a specific programming language (in this case, Smalltalk). Later chapters bring up the topic of constructing algorithms again and again.

While Chap. 2 ignores typical object-orientation concepts, they form the main topic of Chap. 3. At the same time, the chapter also introduces the basic elements of the Smalltalk programming language.

Chapter 4 deals with programming repetitions (“loops”), once again in the context of algorithms. In addition, important methods available in the Smalltalk language for using loops are described.

In order to make it easier for readers to perform practical exercises and to understand the examples provided in the text, Chap. 5 provides instructions for using the VisualWorks development environment, which is used for the programming examples shown in this book.

Chapters 6, 7 and 8 centre on classes, which are the central concept of object orientation. The components of a class definition in Smalltalk are given first, followed by a description of how to create new classes. This is followed by a description of important basic classes in a Smalltalk class library. This leads to the introduction of additional basic concepts of object orientation, such as inheritance and polymorphism.

An entire Chap. 10 is devoted to collection classes, because of their complexity and importance.

¹OLPC means One Laptop per Child.

First though, Chap. 9 summarises information that was already presented in earlier chapters dealing with error messages in the compiler and the runtime system. If—as is likely the case—readers have been confronted with error messages as they reviewed examples in earlier chapters or attempted their own exercises, it might be helpful to read this chapter earlier than its position in this text.

Chapter 11 takes up in a systematic fashion important aspects of object-oriented programming with Smalltalk, some of which, such as blocks and inheritance, already appeared in earlier chapters.

Chapter 12 picks up the principle of recursion, an important aspect of algorithms that cannot be omitted from an introduction to programming.

Chapter 13 deals briefly with processing sequential internal and external streams. It also explains how to access files from Smalltalk programs.

Although space considerations prevent discussion in this book of the development of larger Smalltalk applications, Chap. 14 nevertheless provides a few elementary instructions for how to structure programs. Chapter 17 provides suggestions for sources for this and other topics. The author’s website (<https://brauer.nordakademie.de>) also contains additional information to accompany this book.

Before that, though, Chap. 15 deals with writing component tests and their automatic execution.

The book does not attempt to contrast traditional, procedural development practice with object orientation. Readers who already have programming experience in procedural programming languages are advised to look at literature concerning the Oberon 2 programming language. Reiser and Wirth (1994), for example, provide an excellent description of the transition from imperative to object-oriented programming.

It is important to emphasize once again that no previous knowledge of programming is necessary for successfully working one’s way through this book. It is nevertheless assumed that readers have basic skills in working with a windows-based operating system, such as Microsoft Windows or the Apple Mac OS.

A Comment on Notation

Programming text, to the extent that it is not shown as screen captures, appears as a monospace font. The same is true for various Smalltalk concepts, such as class or method names.

Names of menus and menu contents are shown in **boldface** type.

The Development Environment

The Smalltalk system VisualWorks was used to create the examples in this book. This is a commonly used, professional Smalltalk development environment distributed by Cincom Systems. A fully functional instructional version that runs on many platforms can be downloaded from Cincom’s website, www.cincomsmalltalk.com. Although it is not essential, it will prove very helpful to readers to have access to this system. Screen captures in this book were taken from Version 7.6 of VisualWorks.

Acknowledgements

First of all, I want to thank the publishing company and especially Dr. Klockenbusch, whose engagement is responsible for the appearance of the first edition of this Smalltalk book at a time when the whole world was talking only of a programming language whose name bears a resemblance to coffee. I owe thanks to Sybille Thelen for her support in the publication of this third edition.

I want to thank the employees of the Georg Heeg Company for their critical review of the first draft of this book and for several valuable corrections, suggestions and improvements. I also want to thank my former colleague at the NORDAKADEMIE, Professor Kleuker. Katrin Schimmeyer and Helmut Guttenberg helped me greatly in my struggles toward an orthographically and syntactically correct text. I want to thank Jan Bartelsen for reviewing the third edition.

Elmshorn
August 2008

Johannes Brauer

Addendum to the Preface for the Fourth Edition

For this fourth edition, various small changes and corrections were made throughout the text. In addition, the numerous screen captures for the VisualWorks development environment were updated. For this, I want to thank Cincom Systems, which made Version 7.10 available to me before its release. For her support in this, I want especially to thank Yvonne Schickel.

The most significant improvement, however, is Chap. 16, which deals with an introduction to the development of Web applications in Smalltalk using the framework system, Seaside. The motivation for this chapter rests not least in the fact that the bachelor's degree program for computer science at NORDAKADEMIE is now called "object-oriented development of Web applications."

For reviewing this material and for their valuable suggestions, I am especially grateful to the following people: Stefanie Jasser, Joachim Sauer, Heiko Rehder, Daniel Purrucker and Carsten Becke.

I also wish to thank Bernd Hansemann of Springer Vieweg for his support in the publication of this fourth edition.

Elmshorn
September 2013

Johannes Brauer

Addendum to the Preface for the First English Edition

George Schober from Cincom Systems translated the original text in just a few months. He did an excellent job, and I am very grateful to him.

The numerous screen captures for the VisualWorks development environment were updated again, and for this, I want to thank Cincom Systems who made Version 8.0 available to me prior to its release.

The whole translation project was managed by Yvonne Schickel from Cincom Systems in a proficient and efficient manner. Without her, there would be no English edition. I am deeply grateful to her. Additionally, it's thanks to her, that the book could be extended by Chap. 18, containing ten reports about Smalltalk applications in industry.

I also wish to thank Sybille Thelen and Bernd Hansemann of Springer Vieweg for their support in the publication of this English edition.

Elmshorn
October 2014

Johannes Brauer

Contents

1	Basic Concepts of Computer Science	1
2	Designing Algorithms	7
2.1	Case Study: Currency Conversion	7
2.2	The First Smalltalk Program	8
2.2.1	Entering Program Text	10
2.2.2	Executing Programs	11
2.2.3	Adding Flexibility to the Currency Conversion	14
2.3	Case Study: Solving a Quadratic Equation	16
2.3.1	The Algorithm	17
2.3.2	The Program	18
2.3.3	Generalising the Solving of Quadratic Equations	20
2.4	Summary	27
3	Basics of Object-Oriented Programming Using Smalltalk	31
3.1	Objects, Messages, Methods	32
3.1.1	Messages	36
3.1.2	Case-by-Case Distinctions	39
3.1.3	Blocks	40
3.1.4	Creating Objects—Classes	40
3.2	Literals	45
3.3	Variables and Assignments	49
3.4	Assignment Semantics	53
3.4.1	Using Object Explorer	56
4	Repetitions	59
4.1	Searching for a Maximum Value	60
4.2	Additional Smalltalk Messages for Loops	73
4.2.1	Count Loops	73

4.2.2	Interval Run	74
4.2.3	Collection Run	75
5	The VisualWorks Development Environment	77
5.1	Overview	77
5.2	Starting the Development Environment	79
5.3	Launcher with Transcript	80
5.3.1	Creating One's Own Image	80
5.3.2	Setting System Parameters	81
5.3.3	Using Transcript	83
5.4	Workspace	85
5.5	Inspector	86
5.6	The Debugger	88
5.7	System Browser	88
6	Examining a Sample Class: Circle	97
6.1	Class Hierarchies and Inheritance	97
6.2	Implementing Methods	101
6.3	Alternative Implementation of the Class Circle	108
7	Defining New Classes	113
7.1	Case Study: Currency Conversion	114
7.1.1	Creating a New Class	115
7.1.2	Individualised Class Methods for Creating Instances	121
7.1.3	Defining Instance Methods	124
7.1.4	Expanding the Converter	128
7.2	Case Study: Cinemas	130
7.2.1	Analysis of the Problem Description	130
7.2.2	Implementation	133
7.3	Defining Class Variables	139
8	Class Hierarchies—Inheritance—Polymorphism	143
8.1	The Smalltalk Class Hierarchy	144
8.1.1	Structure	144
8.1.2	Smalltalk's Number Concept	145
8.1.3	The Integer Classes	150
8.1.4	The Classes Float, Double and Fraction	156
8.1.5	Methods Common to All Number Classes	157
8.1.6	Mixed Expressions	161
8.1.7	Truth Values	163
8.1.8	Characters and Character Strings	166
8.1.9	Date and Time	171
8.2	Abstract and Concrete Classes	174
8.3	Generic Methods	177

8.4	Polymorphism	179
8.5	Case Study: Quadratic Equations	180
8.5.1	The Class <code>QuadrEquat</code>	182
8.5.2	Classes for Solution Objects	185
8.5.3	The Solution Methods	189
8.5.4	Examples of Applications	191
8.5.5	Use of Inheritance and Polymorphism	193
8.5.6	Test Programs as Class Methods	193
8.5.7	Error Management	195
9	Debugging Smalltalk Programs	199
9.1	Syntax Errors	199
9.2	Unknown Variables	200
9.3	Unknown Message Selectors	201
9.4	Exceptions	202
9.5	Debugging Methods	203
10	Containers	211
10.1	Unordered Collections	213
10.1.1	The Class <code>Set</code>	213
10.1.2	The Class <code>Bag</code>	216
10.1.3	The Class <code>Dictionary</code>	217
10.2	Ordered Collections	223
10.2.1	The Class <code>Array</code>	226
10.2.2	The Class <code>OrderedCollection</code>	227
10.2.3	The Class <code>SortedCollection</code>	230
10.2.4	The Class <code>Interval</code>	232
10.2.5	The Class <code>String</code>	234
10.2.6	The Class <code>Symbol</code>	234
10.3	Transforming Collections	234
10.4	Case Study: Cinemas	235
10.4.1	Assignment: Display all Cinemas in Transcript	236
10.4.2	Assignment: Display the Profits of a Specific Cinema	243
10.4.3	Assignment: Display all Cinemas Arranged by Size	246
11	Additional Smalltalk Basics	249
11.1	Blocks	249
11.1.1	Blocks as Objects	249
11.1.2	Blocks with Parameters	251
11.1.3	Applications	253
11.1.4	Case Study of a Finite-State Automaton	255
11.2	Inheritance—Method Search	264
11.2.1	Rules for the Method Search	267
11.2.2	The Meaning of the Pseudo-variables <code>self</code> and <code>super</code>	267

11.3	Metaclasses	269
11.4	Object Identity	272
11.4.1	Equality Versus Identity	273
11.4.2	Equality of Objects of Self-Defined Classes	277
11.4.3	Object Copies	279
12	Algorithmic Excursus: Recursion	285
12.1	Recursive Algorithms	286
12.2	Correctness of Recursive Algorithms	293
12.3	Recursive Thinking	295
12.4	Infinite Structures	296
13	Streams and Files	299
13.1	Sequential Access to Ordered Collections	300
13.2	Sequential Access to Files	303
14	Structure of Smalltalk Programs	307
14.1	Standard Method Protocols	308
14.2	The <code>printOn:</code> Framework	311
14.3	Transferring Partial Algorithms to Independent Methods	315
14.4	User Interfaces—The Model-View-Controller Paradigm	321
14.5	Relationships Among Classes	324
14.5.1	Inheritance	324
14.5.2	Association	328
14.5.3	Aggregation	329
15	Systematic Testing	333
15.1	Component Tests	334
15.2	Test Automation Using the SUnit	335
15.2.1	Test Case: Cinemas	335
15.2.2	Additional <code>TestCase</code> Messages	339
15.2.3	An Additional Test for the Class <code>Expenses</code>	340
15.3	Test-Driven Development	341
16	Developing Web Applications	343
16.1	Case Study: Currency Converter	344
16.1.1	Objective	345
16.1.2	The Model for the Currency Converter	346
16.1.3	A First Glance at Seaside	349
16.1.4	Realising the Web Interface	350
16.1.5	Implementing the Functionality	363
16.1.6	Shortcomings in the First Version	366
16.1.7	Refactoring the Method <code>renderFormOn:</code>	366
16.2	Improving the Usability of the Currency Converter	368
16.2.1	Selection Lists in Seaside	370

16.2.2	Checking the Amount Input	371
16.3	Introducing an Administrative Dialogue	375
16.3.1	Creating the Component <code>CcAdministration</code>	375
16.3.2	Calling the Component <code>CcAdministration</code>	377
16.3.3	Seaside's Call/Answer Mechanism	378
16.3.4	Implementing the Administration Dialogue	381
16.4	Incorporating CSS	386
16.4.1	Combining HTML with CSS	389
16.4.2	Defining a <code>style</code> Method	391
16.4.3	Making a CSS File Available in a Seaside File Library	392
16.5	Generalising the Converter	394
16.6	Further Seaside Concepts	399
17	What's Next?	401
18	Smalltalk Applications in Industry	407
18.1	AG5—Safety Compliance	408
18.2	Cognitone— <i>Music</i>	409
18.3	Georg Heeg eK— <i>Software Projects and Services</i>	409
18.4	JPMC— <i>Financial Services</i>	411
18.5	Key Technology— <i>Manufacturing</i>	412
18.6	MetaCase— <i>Software Development Tools</i>	413
18.7	MMA— <i>Insurance</i>	414
18.8	OOCL— <i>Logistics</i>	415
18.9	Rudolph Technologies— <i>Semiconductor Industry</i>	416
18.10	SOOPS— <i>Energy Markets</i>	417
Appendix	Expanding the VisualWorks Image	419
A.1	Adding SunitToo	420
A.2	Adding the Object Explorer	421
A.3	Adding Seaside	421
A.4	The Cincom Public Repository	421
References		423
Index		425

List of Tables

Table 1.1	Applications and problem types	3
Table 8.1	Arithmetic operations for <code>Integer</code> objects	152
Table 8.2	Test operations for numbers	159
Table 8.3	Rounding and truncation	159
Table 8.4	Examples of pattern matching	170
Table 8.5	Important class variables in the class <code>Date</code>	172
Table 10.1	Additional messages for processing the elements of a set	216
Table 10.2	Application examples for the messages in Table 10.1	217
Table 10.3	Messages for accessing components of ordered collections	224
Table 10.4	Application examples for the messages in Table 10.3	224
Table 10.5	Messages for copying ordered collections	225
Table 10.6	Application examples for the messages in Table 10.5	225
Table 10.7	Messages for looping through ordered collections	225
Table 10.8	Messages for adding objects to ordered collections	228
Table 11.1	Messages for sending parameters to blocks	253
Table 11.2	Testing the equality and identity of objects	273
Table 14.1	Standard composition of a protocol for instance methods	310
Table 14.2	Standard composition of a protocol for class methods	311

For many readers, an introduction to computer programming represents their first intensive contact with computer science. For that reason, let's begin by describing a few fundamental concepts that can serve to distinguish programming technology from other facets of computer science. Other texts, such as Ernst (2008), provide a comprehensive introduction to computer science.

Even today the concept *computer science* has not been uniformly defined. Two definitions that one often finds in the literature are:

1. Computer science is the science, technology and application of the processing, storage and transfer of information by machine.
2. Computer science is the science of the automation of human work.

Both definitions taken together are a pretty comprehensive description of what computer scientists concern themselves with. They deal with processing information, which usually occurs for the purpose of transferring to a machine in whole or in part activities that used to be performed by humans.

This machine is called a *computer*, which points to its main initial purpose, the automation of numeric calculations. The German word *Rechner* is equally accurate (to compute = *berechnen*), and is preferable in German. A special characteristic of these machines is that they can be used everywhere, that is, they can be programmed to solve virtually any computational task.

► **Note** Theoretical computer science and mathematics have studied in detail which calculations can and cannot be performed by a computer. This is, however, not the place to look at this in detail.

As everyone today knows, however, computers are not used exclusively—perhaps not even primarily—to perform numeric computations. In private and commercial venues, they

are used to write and format documents of many types, to play games, to call up information on the Internet and for many other purposes. If one looks closely at the internal operation of a computer, one can see that computers at bottom only convert strings of binary symbols¹ into new strings of symbols. Depending on whether one perceives of these strings of symbols as coded numbers or characters (letters, numerals, punctuation), these manipulations of symbols can be numeric computations or the processing of textual information. Using the same trick, one can make computers process images, sounds or video sequences if one first uses binary symbol sequences to encode them.

On the one hand, the fact that programming allows the universal application of computers brings an enormous advantage. On the other, though, it means that one *has to* program computers for every task, no matter how small. What a computer as a “naked” electronic device—that is, as *hardware*—can do makes it useless for the average user. It is only the programs—the *software*—that make a computer a useful tool.

Computer science concerns itself in part with the fabrication of the machines themselves; this part of computer science is called *hardware engineering*. It is closely related to electrical engineering, and especially to microelectronics.

The other part of computer science is *software engineering*, which concerns itself with developing programs. This book is concerned with *programming technique*, which can be viewed as the part of software engineering that concerns itself with basic “mechanical” skills of programming. In contrast, *software engineering* is used to describe the engineering and organisational questions involved in the development of large software projects.

Mayr and Maas (2002) suggest a different subdivision of computer engineering. They look at the first of the two definitions on the first page, and posit the following three areas:

- *Basics*, which come primarily from mathematics, natural sciences, engineering, commerce, and social sciences.
- *IT systems and their development*, which include both hardware and software systems.
- *IT applications*, which involve business engineering, the management of manufacturing processes, medicine, and biotechnology.

Computer science’s main tool for solving problems is thus the computer. Now we will quickly look at how the computer actually proceeds as it executes programs, or rather, how the programs that a computer obeys need to be constructed. The computer performs a human task that needs to be automated according to a firmly prescribed pattern of behaviour laid out in a series of ordered individual steps, which must be strictly adhered to. These patterns of behaviour are called *algorithms*. Generally speaking, algorithms are behavioural instructions for solving problems. In mathematics, for example, we know the Gaussian elimination as an algorithm for the solution of linear equations. Everyday examples of algorithms include recipes or knitting patterns.

¹ Strings of symbols that consist of zeroes and ones.

Table 1.1 Applications and problem types

Field of activity	Type of problem
Administration	Payroll accounting, bank account management
Construction engineering	Structural analysis, numerics
Design	CAD
Manufacturing	Machine tool control
Chemical engineering	Managing chemical reactions
Military engineering	Managing fire escape systems
Operational research	Optimisation of warehousing or transportation systems, management information systems
Traffic engineering	Traffic signal management
Library science	Document identification
Artificial intelligence	Language analysis, automated proof of authenticity

An important characteristic of algorithms lies in the degree to which their formulation is detailed: A complex process is broken down into fundamental steps in such a way that the person who executes the process does not need to know or to understand the problem that is being solved. If one follows the Gaussian elimination exactly, someone who knows the four basic arithmetic calculations can correctly solve a linear equation without knowing what one is. Everyday algorithms like recipes, however, are as a rule not formulated with sufficient precision that an exactly defined result is always produced. Humans who execute algorithms might be able to provide missing information from their knowledge or experience or derive it from the context. But a computer does not really understand anything, and so imprecisions or deficiencies in the formulation of algorithms that they are to process cannot be tolerated.

We can say therefore that an algorithm defines an *automatic solution* to a problem. The (automatic) *execution* of the (problem-solving) algorithm yields the automatic solution of a problem.

The type, extent, and degree of difficulty of algorithms depend not just on the capabilities of the people who develop them, but above all on the type, extent, and difficulty of the problem to be solved, that is, on the field of activity from which the problem arises. (Table 1.1 pairs various fields of activity with the problems that computer science is expected to solve.)

For example, it is easier to have a computer automatically use an algorithm to perform a company's salary calculations than it is to automate the manufacturing processes in the automobile industry or to automatically position millions of transistors on semiconductor chips and connect them by wires.

If machines are to solve problems according to instructions provided to them in algorithms, then the algorithms must be communicated to them in a language that they can understand. Languages that describe algorithms are called *algorithmic languages* or *pro-*

gramming languages. The means of expression in the first algorithmic languages therefore primarily supported programmers in their formulation of algorithms. For example, one of these languages was called ALGOL, an acronym for *algorithmic language*. In modern object-oriented programming languages, such as Java and Smalltalk, this algorithmic aspect has receded somewhat into the background. Nevertheless, since they still serve to program computers, they are still considered to be algorithmic languages.

Programming languages are so-called *formal languages*, which have some things in common with natural languages like French or English. They contain words that can be assembled into sentences according to specific rules, which are called *grammar* or *syntax*. One could say that an algorithm written in a programming language represents a sentence, or a series of sentences, in this formal language. There is, however, a decisive difference between natural and formal languages: A sentence in a formal language that has been correctly constructed according to grammatical rules (called *semantics*) always has only one unambiguous meaning. The meaning of sentences in natural languages can often only be understood from the context or the situation in which they are uttered.

For example, the sentence, “I look to the west” can mean either that the speaker is looking in a particular direction, or that she is expecting assistance or information from nations located in a particular part of the Earth and sharing a roughly similar political outlook.

[Translator’s note: The translation of the original sentence: For example, the sentence, “I look at the west” can mean either that the speaker is looking in a particular direction, or that she is looking for garments in a clothing store. (German *Weste*, pl. *Westen* = vest(s))]

A series of sentences that describe an algorithm are a *program*. Designing an algorithm by means of describing it in the form of a program is an essential part of what is called *programming*.

If an automatic solution to a practical problem is available in the form of a program, and if a specific machine exists that is capable of executing the program, then one can say that the problem has been solved once and for all. Human effort will in principle never again be necessary to solve the same problem.

Information and Data

Computer science can be said therefore to be concerned with the automated processing of information. That means that one speaks of *information processing*. At the same time, though, the expression *data processing* is commonly used. That gives rise to the question, what’s the difference between information and data? At this point we’ll try to answer the question from an information-technology perspective, starting with the way these words are used in everyday speech.

We associate the concept of information with things like

- Accumulated knowledge
- Relevant statements
- Messages

We are especially likely to regard messages as information when they tell us something new, that is, when they transmit knowledge that we did not have before. Information theory, a branch of mathematics, is particularly concerned with the relationship between the “newness” of a message and its informational content; we will not concern ourselves with that any further, though.

Computer science places a premium on distinguishing between the external form (representation) of a message and its meaning (abstract information). Sometimes identical information is presented in different ways, for example, numbers can be presented as Roman or Arabic numerals. The sentence “I look to the west” can be regarded as the representation of a piece of information that can be transmitted acoustically (as a spoken sentence) or optically (as written text). The meaning of this piece of information is independent of the way in which it is transmitted, its representation. The recipient of the message still needs to assign it meaning (interpret it). In order for that to occur, certain prerequisites are necessary. Thus, if the message is available as written text, the recipient must

- Be able to read
- Understand the language in which the text is written
- Have available certain contextual information

Without contextual information the recipient cannot know who “I” is and how the word “west” is intended.

Representations of messages without more have no informational content unless they are interpreted. Independently of the meaning of a message one can also consider its relationship to reality, that is, its validity. That aspect of information is not relevant to computer science.

In summary, we can distinguish three aspects of a message:

- Its representation, that is, its external form
- Its meaning, that is, its abstract informational content
- Its validity

Interpretation completes the transition from representation to information.

Computer science considers *data* to be a representational form of information, usually a structured one. For example, the external form of the dataset

(Doe, John, 123 Main Street, New York, NY)

is interpreted in a particular context in a previously agreed upon manner. In this case, it represents a person’s name and address. The structure of the dataset determines the order of the individual components.

Strictly speaking, a computer does not have the specified capacity to turn data into information, and so it’s more accurate to speak of data processing rather than information processing.

Programs Are Also Data

In fact, computer science does not distinguish between programs and data. The simple text “ $3 + 4$ ” can of course be viewed as a representation of data, as a character string that consists of three characters. At the same time, though, this character string is a Smalltalk program that can be executed by a “Smalltalk engine” (a specially equipped computer, which will be described in detail in Chap. 2). Smalltalk evaluates the text as an arithmetic expression.

It has already been explained that today’s computers can only transform binary character sequences² into other binary sequences. That means that the text “ $3 + 4$ ” must also be transformed into a bit pattern. In order to transform the textual format of this little program, one also needs another program, which the computer can execute only when it is present as binary code in the computer’s RAM. We will return to this topic in Sect. 2.2.

A computer’s RAM contains nothing but bit patterns, which one cannot readily recognise as either programs or data. In order to execute a program, one needs to communicate with the computer to tell it at what point in its RAM the bit pattern of a program begins. A program designer is responsible for ensuring that the program accesses the proper data in the correct bit pattern of the RAM.

²Called bit patterns.

This chapter attempts to explain how one finds an algorithm for a particular problem. Unfortunately, there is no simple answer to this question, and it's utterly impossible to provide a “cookbook” that would allow one to advance with confidence from one problem to the next. Even today software engineering does not have the kind of well-defined design rules that are given in more mature engineering sciences. Instead, there are various codes of procedure that support designers in their work and are meant to increase the probability that the resulting programs will fulfil particular quality criteria, such as *correctness* and *readability*. Section 2.4 provides more information.

One can divide the design of algorithms into two partial sets of tasks. The first step consists of finding a method to solve a particular expression of the problem. This solution method—the actual algorithm—can initially be formulated without regard to what language the machine that will eventually be employed to automatically solve the problem understands. In the next step, though, it is necessary to create a machine-readable version of the algorithm in the form of a program written in a programming language that the selected machine can understand. At first, we'll look at two simple examples in which the first step poses no difficulty. The first example attempts to solve the problem of converting a certain amount of money from one currency into another. In this case, the solution method is obvious. The second example considers the problem of solving quadratic equations, the solution to which is already provided by standard mathematics. In both cases, therefore, we can concentrate on the second step, namely how to formulate the solution in a machine-readable format.

2.1 Case Study: Currency Conversion

Before the introduction of the euro, tourists often travelled across Europe carrying little cards that showed a table for converting amounts from one currency to another. For example, one could convert—with a little effort—amounts from German marks into Austrian

schillings; naturally the exchange rate was not up-to-date. Nowadays when exchange rates can be found online wherever one happens to be, one is likely to reach for one's cell phone.

At this point, we will solve the specific problem of converting a certain amount in Swedish kronor into euros. Assume that the exchange rate is 1 Swedish krona (*sk*) equals 0.108 euros (*eu*).¹ In order to calculate the amount in euros, one can use the following simple formula:

F: $\text{euAmount} = \text{skAmount} \cdot 0.108$

At this point, we need to know what the capabilities of our machine actually are or, more precisely, what expressions in a programming language can make these capabilities available. The only calculation operation we need, multiplication, is naturally part of our machine's repertoire.

One of our machine's peculiarities, though, is that it can only process programs written in the form of a text that contains nothing but characters found on a standard typewriter (or PC) keyboard. Standard keyboards, though, don't contain a multiplication symbol (\cdot); instead we have to use an asterisk (*). Also, we have to be sure that a period is used as the decimal separator, and not another character as is sometimes the case in other countries. Our formula now looks like this:

F: $\text{euAmount} = \text{skAmount} * 0.108$

The "design" of the algorithm for the currency conversion is now complete. It is universally applicable, because it can be used to convert any amount of Swedish kronor into euros. To perform a specific calculation, one needs only to substitute a specific amount for the variable *skAmount*.

The follow section shows how this algorithm can be executed on a real machine.

2.2 The First Smalltalk Program

The machine that we are going to use to execute algorithms will initially be called *SmaViM*, which is an acronym for Smalltalk Virtual Machine. Why did we select this name?

In Chap. 1, it was pointed out that computers are only capable of interacting with binary strings of symbols. That means that programs themselves must be coded in a binary language. The binary code that a computer can directly interpret as an instruction to do something is called a machine language. Because binary codes are very difficult for humans to read, directly programming in a machine language is not merely extraordinarily cumbersome and prone to error, it also manifests the problem that every type of computer uses its own machine language. If one wanted to use a program that had originally been developed for one kind of computer on a different kind, it would first be necessary to translate the code. For that reason, as early as the 1950s a trend started to develop so-called *higher*

¹Exchange rate for May 12, 2008.

or *problem-oriented* programming languages. Well-known examples of such languages include, besides ALGOL, which we already discussed, FORTRAN, COBOL, PASCAL, C and, of course, Java and Smalltalk. These languages were called problem-oriented because the programmer was no longer forced to write instructions in the language of a specific machine, but could instead write problem solutions in a more familiar language, perhaps using mathematical notation. The example of our Formula F shows this clearly.

Translating Programs

In order for programs written in a higher language to still be able to run on a specific computer, they must first be translated into that computer's machine language. This translation procedure can be automated; programs that perform this function are called *compilers*. (The German concept, *Übersetzer*, or translators, is not used very much.) A compiler must in turn exist in the specific machine language so that the machine can run it. One needs a specific compiler for each programming language and each type of machine.

Smalltalk and Java programs operate somewhat differently. In order to avoid having specific computers translate programs that were developed right from the start on different machines, these programs initially define a universal, machine-independent instruction set.² After that sentence occurs, programs are translated only into this quasi-machine language, regardless of which machine they will eventually run on. For each specific machine type, though, one needs a program that can interpret the commands in the quasi-machine language and create for each command a specific binary sequence for the individual machine. This program allows the computer to *seem* to understand the universal instruction set, and for that reason, these machines are called *virtual machines* (VMs). Because of the procedure that was just described, though, they are sometimes also called *interpreters*. SmaViM is thus the VM that can execute Smalltalk programs once they have been translated into byte code.

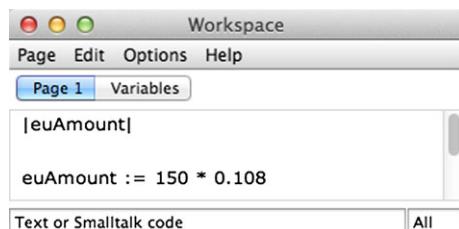
A Smalltalk system has always consisted of not just a programming language and a VM, but also of an integrated development environment. The development environment has a modern GUI, which allows programmers to write Smalltalk programs, as well as to translate and execute them, and also to perform debugging tasks. These Smalltalk development environments are available from several sources; they differ in the type and extent of their components as well as in how they operate. For our purposes, it is sufficient to use the basic functions, which are more or less the same in all development environments. The screen captures used in the following to illustrate how Smalltalk programming works were made using Cincom's *VisualWorks*,³ which is available free-of-charge for instructional purposes.

Chapter 5 describes how to use VisualWorks. Section 5.3.2 explains the required basic settings to enable the student's copy of VisualWorks to resemble the screen captures shown in this book.

²Frequently called a byte code.

³Version 8.0.

Fig. 2.1 Workspace (Mac OS version)



2.2.1 Entering Program Text

Every Smalltalk development environment makes a so-called *workspace* available. The workspace is a window that allows users to key in and edit Smalltalk programming text; it can be used like a window in a conventional text editor. Figure 2.1 shows an example of a workspace. The image represents what one sees when one runs VisualWorks on a Mac OS X platform.

For the sake of comparison, Fig. 2.2 shows the same workspace window under Microsoft Windows 7 OS. In addition, the tools and status-bar options are shown, which can be selected from the **Options** menu. The Mac OS variant will be shown from here on.

The text in the window shows the Smalltalk program for the currency conversion based on the algorithm described in the last section. In contrast to the text in the book, though, the image shows that the variable skAmount has been replaced with the numeral 150. In other words, the program is supposed to determine how many euros 150 Swedish kronor are equal to. In addition, there are a few other small changes, which are due to Smalltalk's syntax.

1. The first line contains, between two vertical lines, the name of the variable used in the program (euAmount). This is a so-called *declaration* of the variables. In general, you must declare variables before you can use them. Between the vertical lines, you can declare as many variables as you like, separated by spaces, tab spaces or carriage

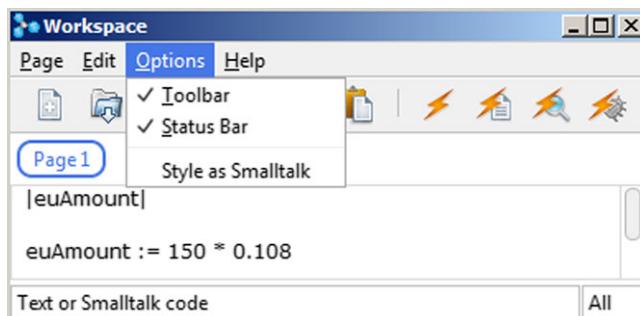
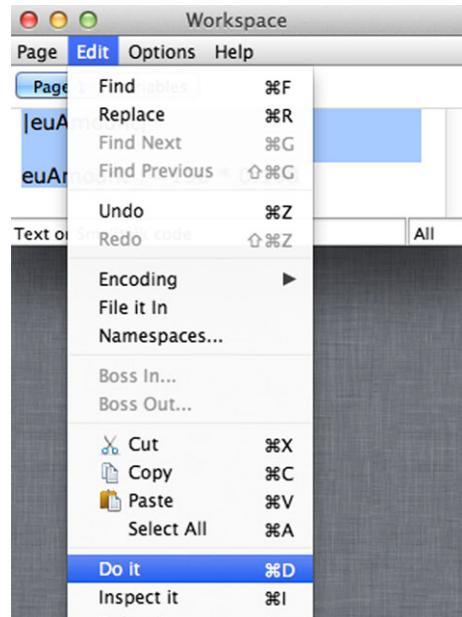


Fig. 2.2 Workspace (Windows version)

Fig. 2.3 The Edit menu in the workspace



returns. More specifically, the variables described here are *temporary* or *local* variables. We will get to know other kinds of variables further on in this manual.

Note for people who know conventional higher programming languages like PASCAL: Smalltalk variables do not have a type. Any random value can therefore be assigned to a variable.

2. In place of “euAmount = . . .” we write here “euAmount := . . .” The character string “:=” expresses the so-called *assignment*. The value of the expression on the right of the equal sign is allocated (assigned) to the variable shown on the left. The equal sign (without the preceding colon) serves in Smalltalk to compare expressions (see Sect. 2.3.3).

2.2.2 Executing Programs

Program text that has been keyed into the workspace can be immediately submitted for execution on SmaViM. First, though, you have to use the left mouse button to select the text. VisualWorks provides two methods for executing the selected text. The first is to left click **Do it** on the dropdown under the **Edit** menu in the workspace (Fig. 2.3).

You can achieve the same effect by clicking **Do it** on the context menu⁴ (see Fig. 2.4).

⁴Open the context menu in Windows by right clicking the selected space. If the Mac OS mouse has only one button available, open the context by pressing **Ctrl** and clicking simultaneously.

Fig. 2.4 Executing a program from the context menu

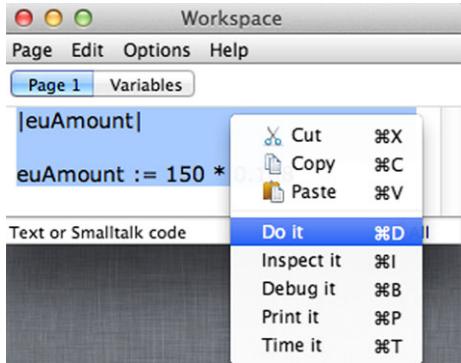
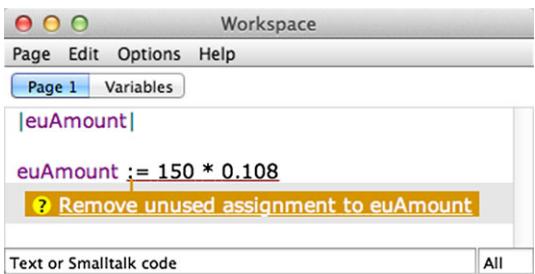


Fig. 2.5 Warning dialog



Whichever method one chooses, the warning dialogue shown in Fig. 2.5 appears, warning the program developer that a value is about to be calculated and assigned to the variable `euAmount`, but that the value assigned to this variable will not be used in the further course of the program. Clicking **remove** would remove the assignment, but that is not what we want at this point. If you hit the **Esc** key on the keyboard, the dialogue disappears.

In spite of the warning, the program is executed and the solution is assigned to the variable `euAmount`, but other steps are necessary before the value can be seen in the workspace. At first, one line is added to the program:

```
| euAmount |
euAmount := 150 * 0.108.
euAmount
```

Since we are now using the variable `euAmount`, the above dialogue will not appear anymore. Note that the second to last line has to be terminated by a full period.

Several methods are available for outputting the values of variables or expressions in SmaViM. The first method is to click **Print it** instead of **Do it** on one of the menus. Figure 2.6 shows the result.

Fig. 2.6 Using Print it to execute a program

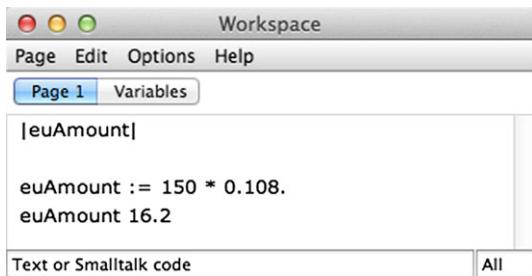
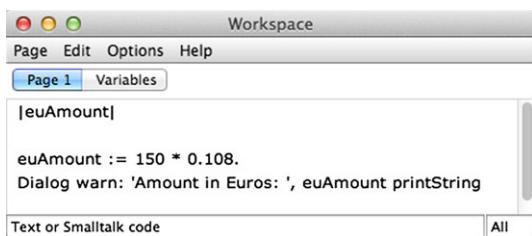


Fig. 2.7 Supplement to the program to display the result in a dialogue window



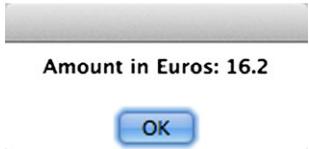
When you use **Print it** to execute a program, the value of the most recently calculated expression will be output in the workspace. Thus, Fig. 2.6 shows the value of the variable `euAmount` next to it.

Another possibility for SmaViM is to display the result in a small dialogue window. That possibility needs to be programmed, however. That is, an instruction to SmaViM must be added to our little program, which opens a dialogue window and displays the results in it. The instruction to do so might look like this:

```
Dialog warn: 'Amount in euros:', euAmount printString
```

At a later point, we will analyse in more detail the form and content of this instruction. To help you understand it, we'll say only this at this point: The instruction `Dialog warn:` opens a dialogue window and displays in it the text that appears after the colon. Texts that are to be displayed in this way as they are written can be any random character string enclosed between single quotes (for example, `'Amount in euros:'`). Numeric values, for example, the value of the variable `euAmount`, will be converted to text only when the command `printString` follows them. The comma between the two expressions serves to combine the two partial texts into one. This is necessary, because the instruction `Dialog warn:` expects a single string of characters as a parameter. Figure 2.7 shows our expanded program. In order to add the output instruction after the first instruction, it must be followed by a period. Stated more simply: two instructions following one another must be separated from one another by a period. If one selects the program text a second time and clicks **Do it**, the dialogue window shown in Fig. 2.8 appears on the screen.

Fig. 2.8 Result output in a dialogue window



And so we have done it; we've solved the problem of converting an amount of currency with a static exchange rate and a specific amount in Swedish kronor. This solution is unsatisfactory, though, in that a change in the rate of exchange or in the amount to be converted requires a program change.

2.2.3 Adding Flexibility to the Currency Conversion

Quite obviously, a program that can convert a specific amount into another currency at an invariable exchange rate is not particularly useful. In order to make the program more flexible, let's first consider a generalised conversion formula:

```
F' : endAmount = startAmount · exchangeRate
```

In this formula, it no longer matters which currencies are involved; all that's necessary is to use the appropriate rate of exchange. The result is the following Smalltalk program:

```
| endAmount startAmount exchangeRate |
endAmount := startAmount * exchangeRate.
Dialog warn: 'Converted amount: ', endAmount printString
```

Nevertheless, SmaViM cannot execute this program because it doesn't know the values for the `startAmount` and `exchangeRate` variables, and consequently cannot calculate their product. In the end, it's up to the user to say which amount is to be converted and at which rate of exchange. This requires instructions that allow the user to enter numeric values during execution of the program, and to allocate those values to the variables `startAmount` and `exchangeRate`. Once again, simple dialogue windows are available for data entry. The instruction

```
Dialog request: 'Amount to convert: ' initialAnswer: '0'
```

produces the dialogue window shown in Fig. 2.9. Any text (enclosed between single quotes) can occur after `request:`; the text appears at the top of the dialogue window (understood in this case as an entry prompt). The field beneath the text serves as a user input field. The value entered after the keyword `initialAnswer:` appears as a default



Fig. 2.9 Data-entry dialogue

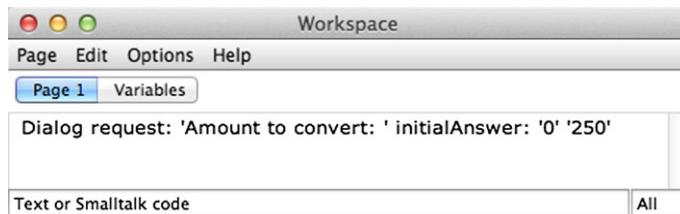


Fig. 2.10 Result of the entry in the dialogue window

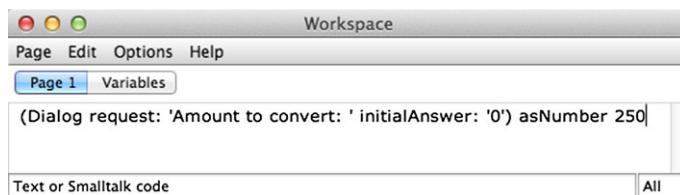


Fig. 2.11 Conversion of the dialogue entry into a number

value in the input field, and will be used as the value if the user enters nothing else. When one uses the command **Print it** to execute the above instruction in the workspace and enters a value of, say, 250 in the input field, once the dialogue entry has been confirmed by clicking the **OK** button, the text '`250`' appears as a result (see Fig. 2.10).

The result of an entry in a dialogue window is always text. Because, however, we can only use numbers to calculate, this text must now be converted into a number; this is, of course, possible only if the entered text can be treated as a number. The exact process involved, that is, which syntactic rules apply for using numbers, is considered in Sect. 8.1.2. The instruction `asNumber` is used to convert text into a number. Figure 2.11 shows both the expanded instruction and the result that appears after using **Print it** to execute the program. This result is now no longer text, but is instead the *number* 250—note the missing single quotes (compare Fig. 2.10).

Numbers entered in this way can now be assigned to the variables `startAmount` and `exchangeRate`. The following partial program accomplishes this:

Fig. 2.12 Conversion program with GUI



```
startAmount := (Dialog request: 'Amount to convert: '
                  initialAnswer: '0') asNumber.
exchangeRate := (Dialog request: 'Exchange rate: '
                  initialAnswer: '1') asNumber
```

Note that the instructions shown here for the dialogue entries wrap over two lines each. This is not required, but is permitted. Spaces and line breaks usually have no meaning in Smalltalk programs.

Now we can complete our program for currency exchange:

```
|endAmount startAmount exchangeRate|
startAmount := (Dialog request: 'Amount to convert: '
                  initialAnswer: '0') asNumber.
exchangeRate := (Dialog request: 'Exchange rate: '
                  initialAnswer: '1') asNumber.
endAmount := startAmount * exchangeRate.
Dialog warn: 'Converted amount: ', endAmount printString
```

This program now lets any amount be converted at any rate of exchange. We should mention here that the process shown here, in which entries are made and displayed using primitive dialogue windows, has nothing to do with the kind of ergonomically acceptable GUI that one expects to find in modern application programs. The result of a “professionally” designed program would look something like Fig. 2.12. Such a program would of course search the Internet for the latest exchange rate and would also be capable of handling values in other units than currencies.

Of course, Smalltalk also allows you to design programs with GUIs that access the Internet or databases. Chapter 16 offers an introduction to those topics.

2.3 Case Study: Solving a Quadratic Equation

This section describes the design of algorithms in a somewhat more complex example. Here we are going to develop a program to solve quadratic equations.

2.3.1 The Algorithm

Let's start by looking at the following quadratic equation:

$$G: x^2 + 2.1x - 5.4 = 0$$

We look for all of G's solutions.

Mathematics teaches us that we can solve G by using the familiar “pq formula”:

$$L: x_{1,2} = -\frac{2.1}{2} \pm \sqrt{\frac{2.1^2}{4} + 5.4}$$

We must expand SmaViM's capabilities if we want to use it to solve this formula. If we had a machine that understood mathematical language (such as L uses), we would be finished, because the formula itself represents the machine-readable formulation of the problem-solving method. Let's assume, though, that SmaViM does not have such an ability. That is not necessary, though, because the formula itself shows how the quadratic equation can be solved using the four basic arithmetic calculations and the derivation of a square root. We can assume therefore that our machine is capable of performing those very operations, and that it can evaluate formulas that contain those operations. (The first case study already showed that the machine could multiply.)

- Addition, subtraction, multiplication and division
- Solving for the square root of a non-negative number
- Evaluation of expressions (formulas) using the operations named above, along with obeying parenthetical ordering (as is usual in mathematics)

In addition, we assume that the machine is capable of solving only one formula at any single moment. The formula L , though, actually contains two formulas for calculating the two solutions to the quadratic equation. For that reason, we note a specific formula for each solution, which the machine will solve in the specified order:

$$x_1 = \frac{-2.1 + \sqrt{2.1 \times 2.1 + 4 \times 5.4}}{2}$$

$$x_2 = \frac{-2.1 - \sqrt{2.1 \times 2.1 + 4 \times 5.4}}{2}$$

Using a simple algebraic redesign, the squaring was replaced by a multiplication.

As already explained in Sect. 2.1, when we create the text for the program, we have to limit ourselves to those characters that can be found on a standard typewriter or PC keyboard. Like the multiplication symbol, the radical and fraction bar are also not available on those keyboards. We can replace the fraction bar as a division sign with the forward

slash (/), and instead of a radical sign, we use the word symbol “sqrt.”⁵ Finally we again replace any decimal commas with decimal points; do not use indexed variables, which would not be possible on a standard keyboard; and again replace the equal sign with “:=.” This results in:

```
x1 := (-2.1 + (2.1*2.1 + 4*5.4) sqrt)/2.
x2 := (-2.1 - (2.1*2.1 + 4*5.4) sqrt)/2
```

Notice that the word symbol `sqrt` is placed after the parenthetical expression from which the square root is to be derived. This writing convention accords with the rules of the Smalltalk language that we will use to write our algorithms.

► **Note for people who know conventional higher programming languages like PASCAL:** These languages customarily use a *functional* notation in which the word symbol `sqrt` (the name of the function) is written in front of the parenthetical expression:

```
x1 := (-2.1 + sqrt(2.1*2.1 + 4*5.4))/2.
x2 := (-2.1 - sqrt(2.1*2.1 + 4*5.4))/2
```

At this point, the algorithm is almost in a format that we could submit for execution to an actual machine. First, though, we should correct a little “blemish.” The algorithm has to derive the same square root twice, which one should avoid. That is because the root is derived in yet another numerical procedure, an algorithm. This requires an expenditure of effort that should not have to unnecessarily occur twice. For that reason, we modify the algorithm so that the value of the square root is derived at the beginning of the program. This interim result becomes a variable that we call `root`, which we use subsequently to calculate the values of `x1` and `x2`:

```
root := (2.1*2.1 + 4*5.4) sqrt.
x1 := (-2.1 + root)/2.
x2 := (2.1 - root)/2
```

2.3.2 The Program

We will now expand the algorithm we wrote in the last section to include variable declarations for `root`, `x1` and `x2`, as well as an output instruction. The program now reads:

⁵“Sqrt” means square root.

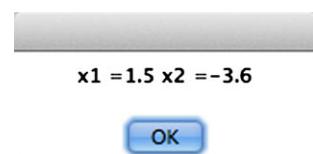
The screenshot shows the SmaViM workspace window. The menu bar includes 'Page', 'Edit', 'Options', and 'Help'. The 'Page' tab is selected. The code area contains the following Smalltalk code:

```
|root x1 x2|
root := ((2.1*2.1) + (4*5.4)) sqrt.
x1 := (-2.1 + root)/2.
x2 := (-2.1 - root)/2.
Dialog warn: 'x1 =', x1 printString, ' x2 =', x2 printString
```

Below the code, there are two buttons: 'Text or Smalltalk code' and 'All'. A vertical scroll bar is visible on the right side of the code area.

Fig. 2.13 Workspace display of a program to solve a quadratic equation

Fig. 2.14 Solution of a quadratic equation



The screenshot shows the SmaViM workspace window again. The code area contains the same Smalltalk code as in Fig. 2.13:

```
|root x1 x2|
root := ((2.1*2.1) + (4*5.4)) sqrt.
x1 := (-2.1 + root)/2.
x2 := (-2.1 - root)/2.
Dialog warn: 'x1 =', x1 printString,
           'x2 =', x2 printString
```

Notice that the three commas in the output instruction combine the four lines of partial text into a single line of text. That is because, as explained above, the instruction `Dialog warn:` expects exactly one text string as a parameter.

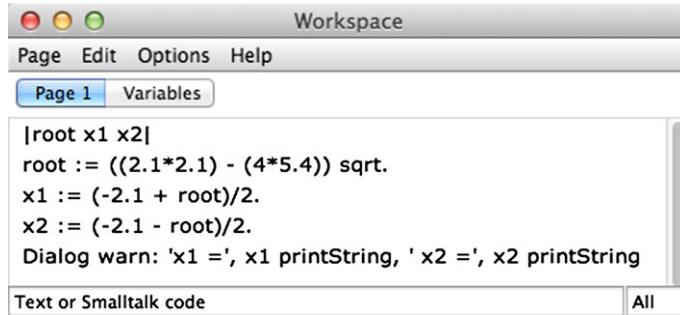
If one enters the programming text in a workspace (as in Fig. 2.13) and then uses the **Do it** command from the **Smalltalk** menu to execute it, the result shown in Fig. 2.14 appears.

And so we've now used our SmaViM machine to solve a quadratic equation.

Naturally we can now change the program so that SmaViM can solve a different equation. But that's an exhausting process that's prone to error, as the following example shows. To solve the quadratic equation

$$G' : x^2 + 2.1x + 5.4 = 0$$

the same procedure we followed above would yield the program shown in Fig. 2.15. If we attempted to execute the program, though, SmaViM would display the error message shown in Fig. 2.16. Since SmaViM cannot derive square roots from negative numbers, the program cannot be used to solve G' .

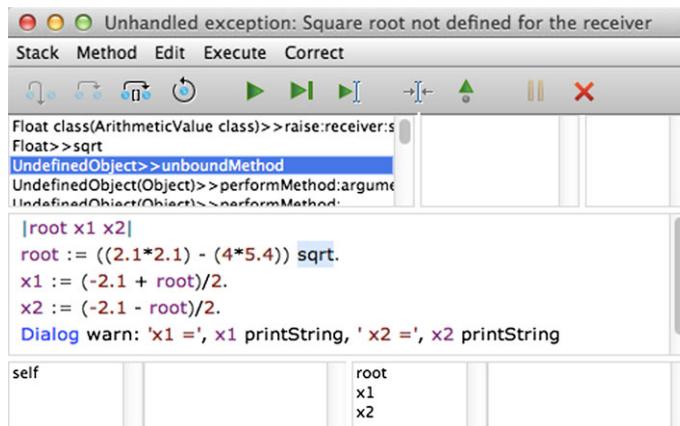


The screenshot shows the SmaViM workspace interface. The menu bar includes Page, Edit, Options, and Help. The tab bar shows Page 1 (selected) and Variables. The code area contains the following Smalltalk-like code:

```
|root x1 x2|
root := ((2.1*2.1) - (4*5.4)) sqrt.
x1 := (-2.1 + root)/2.
x2 := (-2.1 - root)/2.
Dialog warn: 'x1 =', x1 printString, ' x2 =', x2 printString
```

Below the code area, there are two tabs: Text or Smalltalk code (selected) and All.

Fig. 2.15 (Incorrect) program for solving G'



The screenshot shows the SmaViM workspace interface. The menu bar includes Stack, Method, Edit, Execute, and Correct. The code area shows the same code as Fig. 2.15, but the line `root := ((2.1*2.1) - (4*5.4)) sqrt.` is highlighted in red, indicating an error. The status bar at the top displays the message "Unhandled exception: Square root not defined for the receiver".

Fig. 2.16 Error message in SmaViM

In the next section, we will therefore again look at the concept of algorithms and systematically try to figure out how we can generalise the problem of solving quadratic equations in such a way that we do not need to change the program for each new calculation.

2.3.3 Generalising the Solving of Quadratic Equations

Mathematics customarily presents this generalised format for a quadratic equation:

$$Q: ax^2 + bx + c = 0$$

What we are looking for is a method that will solve Q for all real numbers (coefficients) a , b and c .

Using the pq formula on the equation yields this:

$$L_q: \quad x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

L_q cannot be entered into a program in the same way that we solved L in the previous example. What happens, for example, if $a = 0$? We already made reference in the last section to the problem that arises when the radicand (the expression within the radical) might be negative. If we are going to use SmaViM to solve this general equation, that is, to be able automatically to solve for all values of a , b and c , then SmaViM must master the concept of *case-by-case analysis*, because the same formula cannot be used to solve for all values of the coefficients. In other words, SmaViM must be able to make the execution of program steps dependent on conditions.

In order to use case-by-case analysis to automatically solve a problem, you must always be sure that the list of possible cases is complete. That means that, within a program, all cases of partial solutions that arise from the problem result in program steps that are tied to conditions.

In the case where $a = 0$, the equation Q “degenerates” to a linear equation:

$$bx + c = 0$$

It would appear that there is no problem in solving this equation. Taking into account the condition $a = 0$, we write:

```
if a = 0 then x = -c/b
```

In the case $a \neq 0$, the following subcases must be distinguished, which analysis of the radicand ($b^2 - 4ac$) yields:

3. *radicand* ≥ 0 : Using the formula L_q , we get a standard quadratic equation with two real-number solutions. If *radicand* = 0, the two solutions are identical.
4. *radicand* < 0 : There are no solutions in the set of real numbers. We will ignore at this point the possibility of complex numbers.

We could continue to describe the algorithm in the style we started above:

```
if a = 0 then x = -c/b
if a > 0 then x 1 = ...
```

Instead of that, though, we'll write it in a form that uses Smalltalk syntax. Instead of

```
if a = 0 then x = -c/b
```

we write:

```
(a=0) ifTrue: [x := c negated / b]
```

First we write the condition (in this case $a = 0$), enclosed between parentheses, followed by the instruction `ifTrue: [...]`. This means that subsequent instructions written between square brackets (in this case, the single assignment `x := c negated / b`) will be executed if and only if the condition before `ifTrue:` is fulfilled. If the condition is not fulfilled, nothing happens. A series of instructions occurring between square brackets is called a *block*. In Smalltalk, the leading minus sign must be replaced with the word `negated` placed after the variable.

► **Note** Notice the various meanings of the equal sign, which can be used to formulate a condition or to assign a value to a variable (“`:=`”). Note also that Smalltalk programming is case-sensitive, and that the colon in `ifTrue:` is part of the instruction and must not be separated from the alphabetic characters. (In other words, `iftrue:` and `ifTrue:` are both invalid spellings.) In general, though, one can say, somewhat simplifying, that spaces and carriage returns can occur anywhere except within names or instructions.

Let's expand the case-by-case analysis based on the above considerations:

```

1 (a = 0) ifTrue: [x:= c negated / b].
2 (a ~= 0) ifTrue: [
3     radicand := (b * b) - (4 * a * c).
4     radicand >= 0)
5         ifTrue: [
6             root:= radicand sqrt.
7             x1:= (b negated + root) / (2 * a).
8             x2:= (b negated - root) / (2 * a)].
9     (radicand < 0)
10    ifTrue: ["no real-number solution"]]

```

We've numbered the lines of code to simplify the following explanations.

Line 1 deals with the familiar case of a linear equation. In line 2, the character string `~=` means \neq . If $a \neq 0$, we are dealing with an “authentic” quadratic equation, and the subcases described above must be dealt with. This occurs in the `ifTrue:` block, which is part of the condition ($a \neq 0$), which starts with the initial square bracket in line 2 and ends with the final square bracket at the end of line 10. Since the solution of the quadratic equation depends on subcases that are determined by the value of the `radicand`, that value is calculated in line 3 and assigned to the variable `radicand`. Line 4 then checks whether

the value of the radicand is greater than or equal to 0. If that is true (that is, if the value is ≥ 0), the instructions in the associated `ifTrue:` block are executed, beginning with the opening square bracket in line 5 and ending with the final square bracket at the end of line 8.

► **Note** In Smalltalk, the period serves to separate successive instructions. For this reason, a period isn't needed at the end of a series of program steps. The same is true for a series of instructions contained within square brackets. For that reason, there is no period **before** the final square bracket in line 8.

In the event that the value of the radicand is negative—which is checked in line 9—we have initially not programmed a series of instructions. For now, the `ifTrue:` block (line 10) simply contains a comment. Comments are character strings enclosed between double quotation marks that can be inserted anywhere to explain the program text. When the program is executed, SmaViM simply ignores them.

Please notice the nesting of the individual cases. The `ifTrue:` block for the $(a \approx 0)$ condition contains additional `ifTrue:` instructions for the subcases.

A thorough examination of the program, however, makes clear that the very first line already contains a violation of the precept that the case-by-case analysis covers every case. What happens if $b = 0$? The value of the expression `c negated / b` is not defined.

What we obviously need is a systematic list of all relevant cases. Which cases are relevant, that is, those which must be differentiated with regard to the posited solution method, is a question that depends on the problem being solved, and it is not always obvious. In our example, though, we are dealing with a problem sufficiently familiar from mathematics, where it is not particularly difficult to find the relevant cases:

$a = b = c = 0$: This is a trivial case, since every value of x is a possible solution to the equation.

$a = b = 0$ and $c \neq 0$: Because c cannot simultaneously be equal to and not equal to 0, there is a contradiction, and the equation has no solution.

$a = 0$ and $b \neq 0$: This is a linear equation with the solution $x = -c/b$.

$a \neq 0$: Now we have a quadratic equation. With regard to possible real-number solutions, we must proceed to distinguish among the various radicands in the formula L_q , as described above. At the same time, one has to consider a case in which the radicand is equal to 0; in that case, one can avoid deriving the square root of 0. This leads to the following three subcases:

radicand = 0: The solution is $x = -b/2a$

radicand > 0: Two solutions based on formula L_q .

radicand < 0: No real-number solution

This yields the following algorithm to solve Q :

```

(a = 0)
  ifTrue:
    [(b = 0)
      ifTrue:
        [c = 0 ifTrue: ["Trivial solution"].
         c ~= 0 ifTrue: ["Equation unsolvable"]].
      b ~= 0 ifTrue: [x := c negated / b]].

(a ~= 0)
  ifTrue:
    [radicand := b * b - 4 * a * c.
     radicand = 0 ifTrue: [x:= b negated / (2 * a)].
     (radicand > 0)
       ifTrue:
         [root := radicand sqrt.
          x1 := b negated + root / (2 * a).
          x2 := b negated - root / (2 * a)].
     (radicand < 0)
       ifTrue: ["No real-number solution"]]

```

► **Note** For now, in cases where no explicit solution can be calculated, the algorithm contains comments. Lines are indented in the above example to improve legibility of nested distinction of cases. Although SmaViM recognises instructions that are enclosed between square brackets as belonging together, it helps human readers to place alternative cases at the same indented position.

In the event that mutually exclusive cases appear in an algorithm (for example, $c = 0$ and $c \neq 0$), they can be combined in a single alternative in Smalltalk by using the instruction `ifTrue:iffFalse::`. Instead of writing:

```

(c = 0)
  ifTrue: ["Trivial solution"].
c ~= 0
  ifTrue: ["Equation unsolvable"]

```

we can write:

```

(c = 0)
  ifTrue: ["Trivial solution"]
  iffFalse: ["Equation unsolvable"]

```

This not only serves to make the program more readable to humans, but also saves SmaViM the effort of checking two times for the value of the variable c .

If we apply this technique to the entire algorithm, we end up with:

```
(a = 0)
  ifTrue:
    [(b = 0)
      ifTrue:
        [(c = 0)
          ifTrue: ["Trivial solution"]
          iffFalse: ["Contradiction"]]
        ifFalse: [x := c negated / b]]
      iffFalse:
        [radicand:= (b*b) - 4 * a * c].
        (radicand = 0)
          ifTrue: [x := b negated / (2 * a)]
          iffFalse:
            [(radicand > 0)
              ifTrue:
                [root:= radicand sqrt.
                 x1:= b negated + root / (2 * a).
                 x2:= b negated - root / (2 * a)]
              iffFalse: ["No real-number solution"]]
```

Notice here how the case distinctions are nested. Readers should review the program until it is clear to them that—depending on the values for a , b and c —the algorithm always contains only one alternative path, that is, only one `ifTrue: / iffFalse:` branch of the program is executed. To help make this clearer, Fig. 2.17 shows a so-called decision tree.⁶

Each rhomboid node in the tree is a branch where decisions need to be made, until finally exactly one of the actions—shown as squares—must be chosen. One can think of these squares as the leaves of the decision tree.

When constructing a program, it's essential to use indents to make the structure of the algorithm clearly visible, since the structure is primarily determined by case distinctions. One important point is that the instances of the instructions `ifTrue:` and `iffFalse:` belonging to the same alternative should be aligned. At the same time, it's important to remember that the layout of the program text matters only to human readers; it is totally meaningless for SmaViM as it executes a program.

⁶In computer science, the root of a tree is usually shown at the top of a diagram.

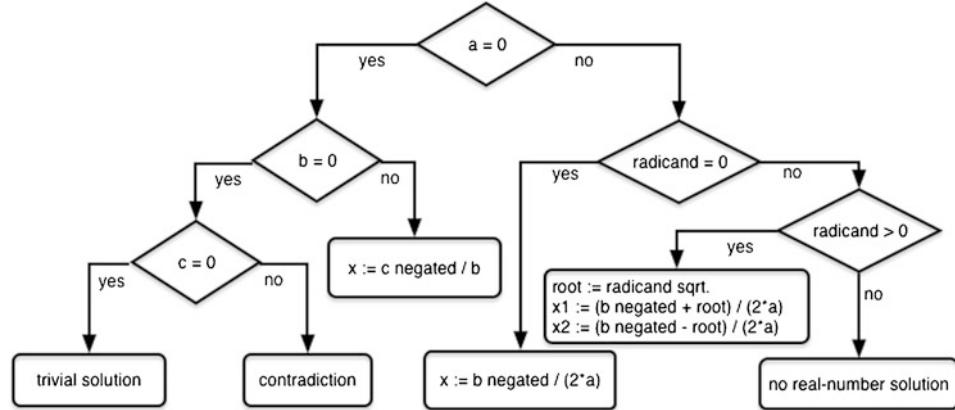


Fig. 2.17 Decision tree

In order to turn the algorithm into a usable program, you must add instructions to provide for the entry of coefficients and the output of results. For this purpose, use the dialogue windows described in Sects. 2.2.2 and 2.2.3. This yields the following program:

```

|a b c x radicand root x1 x2|
a := (Dialog request: 'a=' initialAnswer: '0') asNumber.
b := (Dialog request: 'b=' initialAnswer: '0') asNumber.
c := (Dialog request: 'c=' initialAnswer: '0') asNumber.
(a = 0)
  ifTrue:
    [(b = 0)
      ifTrue:
        [(c = 0)
          ifTrue: [Dialog warn: 'Trivial solution']
          ifFalse: [Dialog warn: 'Contradiction']]
        ifFalse:
          [x := c negated / b.
           Dialog warn: 'x = ', x printString]]
    ifFalse:
      [radicand := (b*b) - 4 * a * c.
       (radicand = 0)
         ifTrue:
           [x := b negated / (2 * a).
            Dialog warn: 'x = ', x printString]
       ifFalse:
         [root := radicand sqrt.
          x1 := (b negated + root) / (2*a)
          x2 := (b negated - root) / (2*a)]]
  ]

```

```
ifFalse:  
  [(radicand > 0)  
   ifTrue:  
     [root := radicand sqrt.  
      x1 := b negated + root / (2 * a).  
      x2 := b negated - root / (2 * a).  
      Dialog warn: 'x1 = ', x1 printString,  
                  'x2 = ', x2 printString]  
   ifFalse:  
     [Dialog warn: 'No real-number solution']]
```

As before, the first line contains the declaration of the required temporary variables.

2.4 Summary

If we contrast the two alternatives for solving a particular problem, using either human or mechanical effort, we reach the following conclusions:

The human must determine the method by which the solution can be discovered. This can occur either by coming upon the solution by analysing the problem, that is, by thinking about it, or else by finding an existing solution to the problem, either by researching it in a relevant text or by remembering that one had already solved the problem at an earlier date. At bottom, though, a human has to discover the method at least once.

If it's a familiar method, the concept behind the solution can be applied to the concrete problem.

The machine, on the other hand, doesn't have to bother with a method. A program supplies the machine with the method in the form of a program. Once the machine has the program, it can solve the problem by applying the method to supplied values. It executes the program.

In other words, a human is assigned, in the form of a problem description, the task of solving a problem. The description establishes what is to be accomplished. The way the problem is solved does not, in principle, matter to the person who assigns the problem. If the solution to the problem has an economic value, the only side issues are that the solution should be found as quickly as possible and that it require the work of as few people as possible. It should cost as little effort as possible.

The machine doesn't know what kind of a problem it is solving. The program tells the machine exactly what it needs to do.

On an economic level, therefore, it pays to employ a machine if its purchase and subsequent upkeep during its working life cost less than the labour it replaces by means of its capabilities, that is, its mechanical functioning. That will always be true when the problem solutions (program executions):

- Are required repeatedly (for example, payroll accounting)
- Require intensive calculation (for example, construction statics)

Machines also continue to conquer new fields of problem resolution, based on the following situations:

- Machines are fast and secure. They make possible, for example, the solution to problems involving space travel (travel to the moon and landing on it), military engineering and weather prediction.
- At base, machines like modern computers have universal applicability. In principle, they can solve any problem that's capable of a mechanical solution.

This final ability—computers' ability to solve all mechanically solvable problems—is called their programmability. It means that a machine can, in principle, master every problem-solving method that can be put into machine-solvable terms based on the computer's design and the mechanical functioning attendant on its design. In other words, a machine can execute any program. SmaViM also has that ability.

How Does One Find an Algorithm?

Back at the beginning of the chapter we discussed the difficulties involved in finding a correct method to solve a given problem and in turning that method into an error-free program. Computer science has so far been unable to develop a methodology for this process that can guarantee error-free programs. Anyone who spends time working with computers realises that software does not always satisfy the demands put on it, nor does it always function correctly. To name just one spectacular example, the military has been known to explode rockets shortly after take-off because a software error would have caused an even worse catastrophe. One should always keep such incidents in mind and carefully examine the use of computers, especially in cases where human safety depends on the correct functioning of a computer-controlled system.

One of the processes that can support programmers as they develop algorithms is so-called *step-by-step refinement* (Wirth 1971). An entire task is broken down into partial tasks. The partial tasks are in turn broken down into even smaller tasks until their solutions can be expressed in simple terms in the selected programming language. In Sect. 4.1, we will return to this process as we work through an example. Object-oriented software design uses a similar method, which is called the *composed-method* model (Beck 1997); we will discuss it in Sect. 14.3.

Separating Algorithms and User Interaction

The final version of the program used to solve quadratic equations, presented at the end of Sect. 2.3.3, contains both instructions that serve the actual calculation of solutions as well as others that accomplish interactivity with the user, that is, the instructions that allow the user to enter data or view results. The two types of instruction are only tangentially related

to one another. The solution method is independent of the source of the entries or what happens to the calculations after they have finished. There is a great deal to be said for separating the two program components. That would allow the solution process to be used in another context, where perhaps the input data is supplied by another part of the program and the results are processed by yet another, requiring user action in neither case.

The following chapters concentrate primarily on converting solution processes using the medium of an object-oriented programming language. The topic of user interactivity does not play a role until Chap. 16. Chapter 7 presents a version of the quadratic-equation program that does not contain the input/output instructions. Section 14.4 and Chap. 16 both discuss the separation of problem solution and user interactivity.

Basics of Object-Oriented Programming Using Smalltalk

3

One might condense Smalltalk’s most notable characteristic into the sentence, “Programs consist exclusively of *messages* that are sent to *objects*.” This names two of the basic concepts of object-oriented programming, whose importance for the Smalltalk programming language will be dealt with in detail in Sect. 3.1. Before going into detail, though, we should first look at a few introductory considerations concerning our motivation for using an object-oriented programming style.

Naturally we notice that the word *object* keeps popping up. But what do software engineers mean when they use that term? If you look around you, you can see many objects from the real world. For example, there are things that we can see and hold onto: a car, a television, a piece of furniture. But sometimes we also call a cat or a person—maybe lacking a little respect—an object. These objects all have something in common; they possess a state of being made up of certain feature characteristics, and they all have a certain behaviour. A housecat’s state of being could be determined, for example, by its name, the pattern of its coat and how awake it is. Its behaviour consists of eating, hunting, pricking up its ears and so on. A car’s state of being might be described by its type of transmission, the number of wheels and its speed at any given moment, while its behaviour could be characterised by its acceleration, braking behaviour and how it changes direction. But objects can also be of a more abstract nature, for example, events, amounts of money or contracts of sale.

An essential task in software development consists of reconstructing, to some extent, the real world inside of a computer, in that the state and behaviour of real objects are replicated by means of programs. We call this image of the real world a *model*, and the process of creating it we call *modelling* or *model construction*. It seems illuminating doesn’t it, that a programming language that uses the concept of objects, which is based on concrete objects, simplifies this reconstruction of reality? Object-oriented programming languages may be said to recognise software objects that also possess a state of being and a behaviour. The next section explains exactly how this is handled technically.

An import aspect of model construction is *abstraction*. The image of a real-world object in a computer will usual display fewer characteristics than the real-world object itself. The totality of the identifying characteristics of all housecats is so manifold that they can hardly be encompassed within a model. The behaviour of a living organism is also much too complex to be capable of being technically completely reproduced. How a cat reacts to certain external stimuli can neither be exactly predicted nor is it even completely determined.

The exact reproduction of real-world objects is for that reason frequently impossible, but it is also not really necessary. The modelling always occurs in front of the background of a specific application program, so that it is possible to *abstract* from those characteristics of real-world objects that are not important for a particular application. When trucks are to be modelled for the management of a fleet, the characteristic *speed at a given moment* is unimportant, as are behaviours like *steering* or *accelerating*. But if, in contrast, one is developing steering software for a driving simulator, those particular object characteristics cannot be abstracted.

Abstraction is an important principle that in general plays an important part in human society. Just think of how much the process of making payments has changed. The transition from bartering to paying with money was a significant process of abstraction. Cashless payment represents yet another level of abstraction, which has been able to achieve its importance in the world today because of a technical innovation. Certain characteristics of a given amount of cash—for example, the fact that it can be broken down into particular coins or bills—is no longer important.

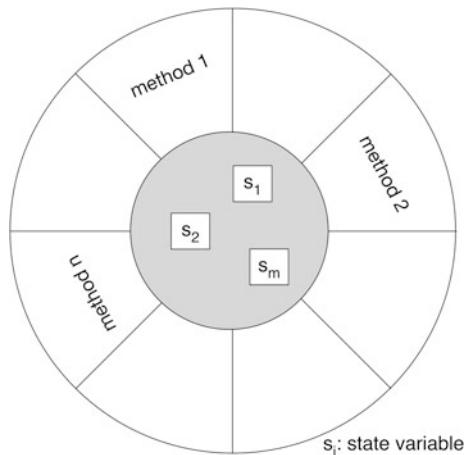
The automation of human work is always connected with the construction of models, and so an ability to abstract is one of the most important abilities that can be demanded of a software developer. Object-oriented programming should support programmers by meeting them halfway and offering them, in the form of software objects, a powerful modelling medium. This gives hope to the possibility that the goal of creating—in an economical way—software that meets requirements can be achieved more easily than with other programming styles.

In object-oriented programming, a program is viewed as an ordered series of messages directed toward a goal, which are sent to appropriate objects. The way the objects behave when they receive the messages can take the form of a change in the status of the objects, or else in further transmission of messages to other objects.

3.1 Objects, Messages, Methods

Object-oriented programming uses a different vocabulary to describe concepts and techniques that are in large part already familiar from conventional programming methods. Nevertheless, we will first describe here the basic concepts of object orientation, which were introduced by Smalltalk before most other object-oriented programming languages adopted them.

Now, in Chap. 2, we introduced a complete Smalltalk program without once mentioning either *objects* or *messages*. Both the currency converter and the program that solved

Fig. 3.1 Object structure

quadratic equations actually only used numbers. One of Smalltalk's typical characteristics is that it treats numbers as objects. At first glance, that may seem surprising, since we don't typically think of numbers as exhibiting behaviour. Nevertheless, Smalltalk is very consistent in observing object orientation. Everything is an object, including numbers. What it means for such numeric objects to exhibit behaviour will be made clear during the course of this section.

Before we take another look—this time from the perspective of object orientation—at the programs developed in Chap. 2, we will here consider more deeply the “nature” of objects in the sense of object-oriented programming. An object consists of two parts:

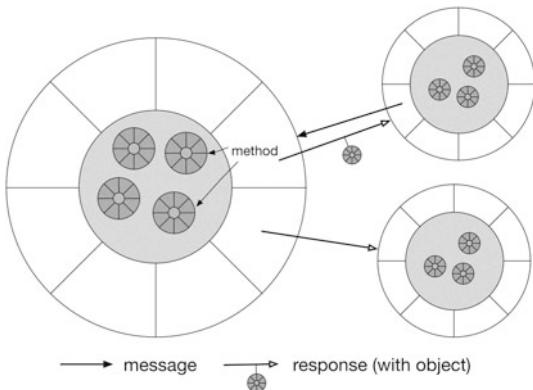
- The *status* of the object, represented by its data structure
- The *behaviour* of the object, which establishes the manner in which the object reacts when it receives a message

We can say that, as an object-oriented program runs, objects exchange messages. Possible ways that an object might react when it receives a message include:

- The object provides information about its status.
- The object changes its status.

As a fundamental principle, the status of an object is not visible from the outside; only the sending of messages makes it visible or allows it to be changed. This is a characteristic of objects called *information hiding*, which is one of the most important characteristics of object orientation. Figure 3.1 illustrates this principle. For every message that an object understands, it has at its disposal a so-called *method*,¹ in which the reaction of the object

¹Comparable to a procedure in a standard, imperative programming language.

Fig. 3.2 Message exchange

to a particular message is established (programmed). We can say that the methods form the externally visible surface—the *interface*—of the object. The data structure, formed from the state variables, is the core of the object, which is invisible to the outside. The state variables are also called internal variables or instance variables and serve to display the objects that represent the internal status of the object.

The execution of a method as a reaction to the receipt of a message consists in turn in sending additional messages to other objects, including those that are represented by the internal variables. In the end, one object is always returned as a result to the sender of the message. Figure 3.2 illustrates this process. In this image, the internal variables of the objects are also shown as objects (with interface and core). To make it easier to see, only a single response is shown, even though each time a message is sent a response is returned.

► **Note on the terminology:** In Chap. 2, we used the word *method* in its colloquial sense as a synonym for procedure, process, algorithm, etc. But now we've introduced a special concept, which has a specific meaning in the context of object-oriented programming. In order to avoid confusion about which meaning of method we have in mind, the rest of this book will use it only in the specific, object-oriented sense. If we intend to use the colloquial meaning of the term, we'll use the term *procedure*. As explained above, sending a message results in the execution of a method. This process is called *method activation* or, to borrow the term procedure call from procedural programming, the process is sometimes called a *method call*.

Let's look again at portions of the programs we developed in Chap. 2. For example, arithmetic expressions of the following type occurred:

```
((2.1 * 2.1) + (4 * 5.4)) sqrt
```

In object-oriented terminology, one reads the partial expression `4 * 5.4` in this way: The message “`*` 5.4” is sent to the object “4”. The receiving object “4” responds to the receipt of the message with the result of the multiplication, the object “21.6”. Of course, “5.4” is also an object, which in this case is called an *argument*² and is a part of the message “`*` 5.4”. The symbol “`*`” represents here the so-called *message selector* (or just selector), which tells the recipient of the message what kind of operation is to be carried out (multiplication in this case), that is, which method of the receiving object is to be used. We can say that the message transmits to the object “4” the command, “Multiply your value by 5.4 and respond with the result of the multiplication.”

► **Note:** In Smalltalk, a significant characteristic of the way objects react to the receipt of messages is that the receiving object always sends back a response in the form of an object.

Let’s look now at how SmaViM processes the entire expression

```
( (2.1 * 2.1) + (4 * 5.4)) sqrt
```

When Smalltalk encounters a compound message expression, it always processes the component messages from left to right, unless parenthetical expressions force it to process the expression in a different order. With respect to the message selectors “`*`” and “`+`”, representing arithmetic operations, there are also no rules of precedence along the lines of “multiply and divide before you add or subtract”. This leads us to the following execution sequence for the expression:

1. The message “`*` 2.1” is sent to the object “2.1”. The object responds with the result object “4.41”. One can now think of a restated expression `(4.41 + (4 * 5.4)) sqrt`.
2. As dictated by the placement of the parentheses, the next operation involves sending the message “`*` 5.4” to the object “4”, which responds with the object “21.6”. The expression now has the format `(4.41 + 21.6) sqrt`.
3. Now the object “4.41” becomes the recipient of the message “`+` 21.6”. And the response object is “26.01”.
4. The resulting expression is now `26.01 sqrt`. The message selector in this case is “`sqrt`”. The receiving object is now commanded to calculate the square root of its value and to respond with the result.
5. The object “5.1” represents the final result ($\sqrt{26.01} = 5.1$).

²The term *parameter* is also used sometimes.

3.1.1 Messages

The basic format for a message expression in Smalltalk is

```
<object> <message>
```

in which `<object>` signifies the receiving object of the message `<message>`. Remember, though, that the receiving object can also be the result of a message expression. In the example described above, the response object of the expression “`2.1 * 2.1`” is the recipient of the message “`+(4 * 5.4)`”.

As we’ve already seen in the example, some messages need no arguments. Such messages are called *unary messages*. “`sqrt`” is an example of a unary message. Messages that require exactly one argument (for example, “`*`” or “`+`”) are called *binary messages*.

The following are additional examples of unary messages:

<code>2 negated</code>	Result: <code>-2</code>
<code>25 factorial</code>	Result: <code>25! = 15511210043330985984000000</code>
<code>Window new</code>	Result: A new window on the screen
<code>alpha sin</code>	Result: <i>sin alpha</i> , where <i>alpha</i> must be a variable to which a numerical value was assigned, interpreted as an angle in a radian.

The significant characteristic of unary messages is that they are always represented by a single word symbol, for example, *negated*.

Binary messages in Smalltalk include the arithmetic operations addition, subtraction, division and multiplication (`+, -, /, *`). They also include the so-called comparative operations (`=, <, >, <=, >=, ~=`). We encountered these numerous times in the solution to the quadratic equation, for example, when we tested whether one of the coefficients was equal to 0:

```
(a = 0) ifTrue: [x := c negated / b]
```

The message “`= 0`” means therefore that one is to test the receiving object “`a`” to see if its value is equal to 0. The object responds with a yes or a no. More precisely, the response object is either `true` or `false`. `true` and `false` are considered so-called *pseudo-variables*, the value of which cannot be changed. The pseudo-variable `true` represents the logical value “true”, while `false` represents the logical value “false”. `True` and `false` are also sometimes called Boolean objects.

The argument of a binary message can be any message expression whose solution is an object that eventually, upon solution, becomes the argument of the message. In the expression `4.41 + (4 * 5.4)`, the expression within parentheses is the argument of the message

“+”. As has already been explained, this expression is solved, and the result object is passed on to the receiving object “4.41” with the message “+”.

Additional binary messages will be dealt with in later chapters. The significant characteristic of binary messages is that they are always represented by simple (for example, +) or compound (for example, <=) operands.

The third and last type of message is the *keyword message*. Keywords are simple word symbols with an attached colon.

In

```
(a = 0) ifTrue: [x := c negated / b]
```

`ifTrue:` is such a keyword. An argument is associated with each keyword. In this case, it is a block (text between the square brackets). In this example, the keyword message is

```
ifTrue: [x := c negated / b]
```

The receiving object is the Boolean object that results from the solution of `(a = 0)`. Chapter 2 has already described how the keyword “`ifTrue:`” operates, and it will be described in greater detail in Sect. 3.1.2.

A further simple example of the application of a keyword message is

```
a max: b
```

Assuming that the two variables represent objects that can be compared on the basis of their size, numbers, for example, this expression produces the result object `a` if `a` is greater than `b`; otherwise it produces `b`.

It is important to note that keyword messages can consist of multiple keywords. We’ve already encountered the message `ifTrue:ifFalse:`. Associated with each keyword is an argument, which is written after the keyword’s colon. An example from Chap. 2 is

```
[c = 0
  ifTrue: ["Trivial solution"]
  ifFalse: ["Contradiction"]]
```

There is no limit on the number of keywords that can be in a keyword message in Smalltalk.

We still have to explain the order in which messages are sent to objects when a complex message expression contains several messages of varying types. The following rules always apply:

1. Messages within parentheses are always evaluated first.
2. Unary messages are evaluated before binary ones.
3. Binary messages are evaluated before keyword messages.
4. If an expression contains multiple messages of the same type, they are always processed from left to right.

Examples

- In `3 + 4 sqrt`, Rule Number 2 dictates that the square root of 4 is taken and then added to 3.
- In `7 + 2 * 3`, Rule Number 4 dictates that the first two numbers are first added together, and their sum is then multiplied by the third number.
- In `6 max: 3 + 4`, Rule Number 3 dictates that `3 + 4` are added together before the message `max:` is sent to the number 6.
- The following table shows in detail the evaluation of the expression

`2 + 4 * (6 negated max: 8 - 3) negated`

The left column shows in boldface type the next message to be evaluated within the expression. The middle column shows the result of the evaluation of the message. In the next line of the table, this result appears in place of the original message. The third column indicates the rule(s) used to determine the precedence of the various messages.

Next message to be evaluated	Result	Rule(s)
<code>2 + 4 * (6 negated max: 8 - 3) negated</code>	-6	1 and 2
<code>2 + 4 * (-6 max: 8 - 3) negated</code>	5	3
<code>2 + 4 * (-6 max: 5) negated</code>	5	1
<code>2 + 4 * 5 negated</code>	-5	2
<code>2 + 4 * -5</code>	6	4
<code>6 * -5</code>	-30	-

Difficulty arises if multiple keyword messages appear in an expression. Assume that we want to compare the greater of a variable `z` and the lesser of the variables `x` and `y`. We might naively write the expression this way:

```
x min: y max: z
```

Nevertheless, the assumption that the messages `min:` and `max:` are to be read from left to right is incorrect. If we try to execute the expression in the workspace, the error message shown in Fig. 3.3 appears. That is because SmaViM is not able to distinguish between the sequential execution of two keyword messages that each consists of a single keyword (in this case, `min:` and `max:`) and a keyword message that consists of two keywords (in this

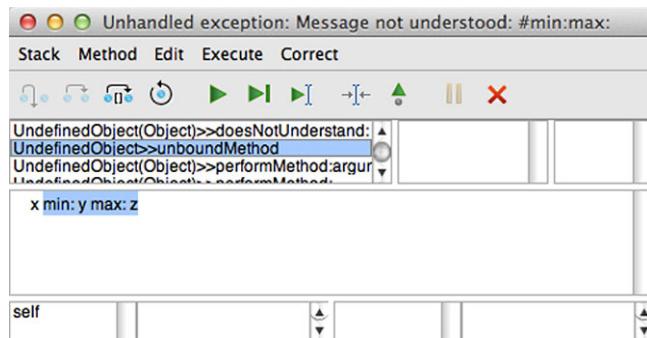


Fig. 3.3 Error: Recipient does not understand the message `min : max :`

case, `min : max :`). For that reason, sequentially occurring keyword messages must always be enclosed between parentheses. The example shown above must therefore be written in the format:

```
(x min: y) max: z
```

3.1.2 Case-by-Case Distinctions

Case-by-case distinctions in Smalltalk are programmed using one of the four messages `ifTrue:, ifFalse:, ifTrue:ifFalse: and ifFalse:ifTrue:`, which have already been used in the preceding examples.

► **Note for people who know conventional higher programming languages like PASCAL:** These languages usually have unique command types with special syntax to process case-by-case distinctions (for example, the if-then-else command in PASCAL). Smalltalk, in contrast, realises case-by-case distinctions as ordinary keyword messages.

These messages are understood only by the two Boolean objects that are represented by the pseudo-variables `true` and `false`. Let's look again at the example

```
(a = 0) ifTrue: [x := c negated / b]
```

Somewhat more precisely than we have before, we might describe the behaviour of this expression in the following way: If `true` is the receiving object of `ifTrue:`, then it responds with the object that results from the evaluation of the expression between square

brackets. If, on the other hand, the receiving object is `false`, then it ignores the text transmitted as a parameter (that is, the parameter is not evaluated) and answers with the undefined object that is represented by the pseudo-variable `nil`. The behaviour of the messages `ifFalse:`, `ifTrue:ifFalse:` and `ifFalse:ifTrue:` can be determined in analogous ways.

3.1.3 Blocks

The arguments for messages in case-by-case distinctions are blocks. A block is a sequence of messages enclosed between square brackets, where the messages are not immediately evaluated. As described in the preceding section, only a receiving object can determine whether the block of an `ifTrue:` message may be evaluated. That means that blocks are always used in cases where sequences of messages are to be executed only under certain conditions or—as we will see later—repeatedly.

On the other hand, like everything in Smalltalk, blocks are objects that can be associated with a variable and to which one can send messages. A block especially understands the message `value`, which causes it to process the series of messages that it contains. Let's look at the following instructions:

```
x := 0.  
aBlock := [x := 7].  
aBlock value.
```

In the second line, the block `[x := 7]` is associated with the variable `aBlock`. However, since the instructions in this block are not executed, the variable `x` keeps the value 0. In the third line, the message `value` is sent to the block. The instructions in the block are then executed and the variable `x` receives the value 7.

Subsequent chapters describe other applications for blocks.

3.1.4 Creating Objects—Classes

Up until now, we have gotten to know three kinds of objects: numbers, Boolean objects represented by the pseudo-variables `true` and `false` and character strings that until now we have used only to display text in dialogue windows. In the meantime, we have also learned how to send messages to objects. We have occasionally referred to the fact that an object understands particular messages. For example, numbers understand the messages “+” and `negated`. But it obviously makes no sense to send a Boolean object the message `sqrt` or to try to multiply two character strings. In other words, we must consider the question, which objects can understand which messages?

Object-oriented programming groups “like” objects into so-called *classes*. Objects of a single class display the same internal structure and understand the same messages. Each class has a name that by convention starts with an uppercase letter. For example, the class of whole numbers bears the name `Integer`; the class `String` stands for character strings. The internal structure of objects of the class `Integer` consists of a single numerical value, while the structure of objects of the class `String` consists of the individual characters that form the character strings.

Which messages an object understands depends on which class it belongs to. Each message in a class definition has a method available; that is, for each message selector that the objects of a class understand, there is a method, whose name agrees with that of the message selector. The list of all methods within a class is also referred to as the class’s *method protocol*.

It is entirely possible that objects of different classes might accept the same message selector, but still execute unique methods. For example, both whole numbers and character strings can be compared with one another using the message “`<`”, as the following examples show:

```
3 < 4 Print it produces: true  
'Carl' < 'John' Print it produces: true  
'Sally' < 'Phyllis' Print it produces: false
```

The method protocol for both classes contains the message selector “`<`”, but different methods are defined for it in each, since whole numbers are compared in a different way than Strings.³

But there are also messages that `Strings` can understand that whole numbers cannot. Character strings can be combined into longer strings using the message “`,`”:

```
'Carl', 'and', 'Phyllis' Print it produces 'Carl and  
Phyllis.'
```

If one tried the same sort of thing with numbers, SmaViM would answer with the error message that the recipient does not understand the message (see Fig. 3.4).

Now let’s turn to the question: How are objects actually created? In the previous examples, we’ve mostly created new objects simply by “writing them down.” That’s been especially true with numbers and character strings. Constants written in this way are sometimes called *literals*. But there are only a very few classes in which we can create instances by writing literals (see also Sect. 3.2).

In most cases, objects are created by messages that are sent to the relevant class. This is possible in Smalltalk because classes are also objects that have methods at their disposal. In general, these are referred to as *class methods*.

As an example, let’s consider the class `Rectangle`, the objects of which are four-sided figures with four right angles in a two-dimensional Cartesian coordinate system. A rectangle is determined by specifying the end points of a diagonal. The simplest method

³Stated simply, one string is smaller than another when it occurs earlier in an alphabetic list.

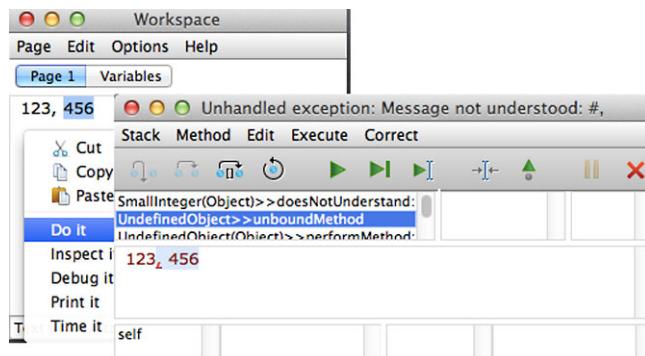
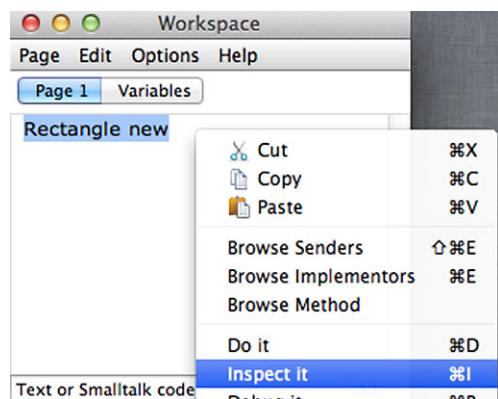


Fig. 3.4 Error: Recipient does understand the message “,”

Fig. 3.5 Creating and Inspecting a new rectangle



for creating an object in the class `Rectangle` is to send it the message `new`. If one executes an appropriate expression in the workspace—as shown in Fig. 3.5—using the **Inspect it** command, then VisualWorks starts a so-called *Inspector* that enables one to examine the inside of an object. The result can be seen in Fig. 3.6. The window name in Inspector indicates that the object in view is an instance of the class `Rectangle`. The left-hand pane describes the structure of a `Rectangle` object.⁴ As mentioned above, a rectangle is determined by the endpoints of its diagonals; the endpoints are called the `origin` and the `corner`. These are the instance variables for the class `Rectangle`.

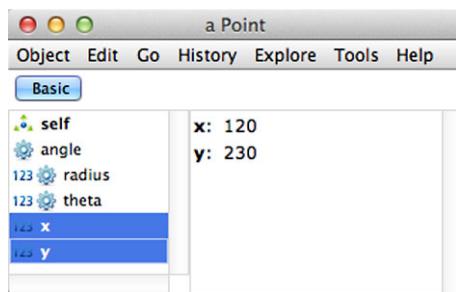
Selecting an instance variable in the left-hand pane of the Inspector window causes its value to be displayed in the right-hand pane. The value is always an object. In Fig. 3.6, the value of the `origin` variable appears as the so-called *undefined object*, which is represented by the pseudo-variable `nil`. This is because the message `new` does not permit any entries concerning the dimensions of the rectangle to be created.

⁴For now, we will ignore the line labeled “self”.

Fig. 3.6 Inspector for an object of the class Rectangle



Fig. 3.7 A Point object



For a “correct” rectangle, though, the two instance variables must represent two points. Points in Smalltalk are represented as examples of the class Point. A Point object represents a pair of numbers, one of which represents the x-coordinate and the other the y-coordinate of the point. Customarily, the origin of the coordinate system is the upper left corner of the screen, window or other graphic context; x values increase toward the right and y values increase toward the lower part of the field. One can create an instance of the class Point in a variety of ways. One possibility is to send the binary message “@” to a number, using another number as an argument. Evaluating the expression

```
120 @ 230
```

produces a Point object with an x coordinate of 120 and a y coordinate of 230.

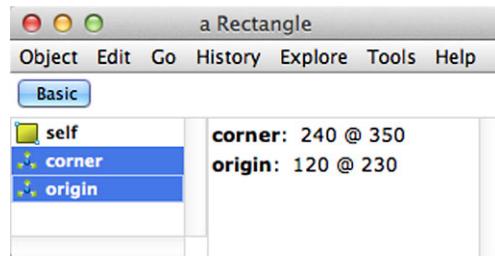
Running **Inspect it** in the workspace displays the Inspector shown in Fig. 3.7. One can see that examples of the class Point use the instance variables x and y.

It is also possible to create a point by using the keyword message `x:y:`, which is sent to the class Point. The class Point thus owns a class method with the same name. The expression

```
Point x: 120 y: 230
```

displays the same point as the previously described method.

Fig. 3.8 A Rectangle object with defined corner points



Now we are able to create a rectangle by actually specifying Point objects as endpoints of the diagonals. To do this, there are again various alternatives. One can send the message `corner:` to a Point object using another point as the argument. Using **Inspect it** to evaluate the expression

```
120 @ 230 corner: 240 @ 350
```

displays the Rectangle object shown in Fig. 3.8. The image shows both the instance variables `corner` and `origin`, the values of which are Point objects.

One can also create the same rectangle using the class method `origin:corner:` from the class Rectangle:

```
Rectangle origin: 120 @ 230 corner: 240 @ 350
```

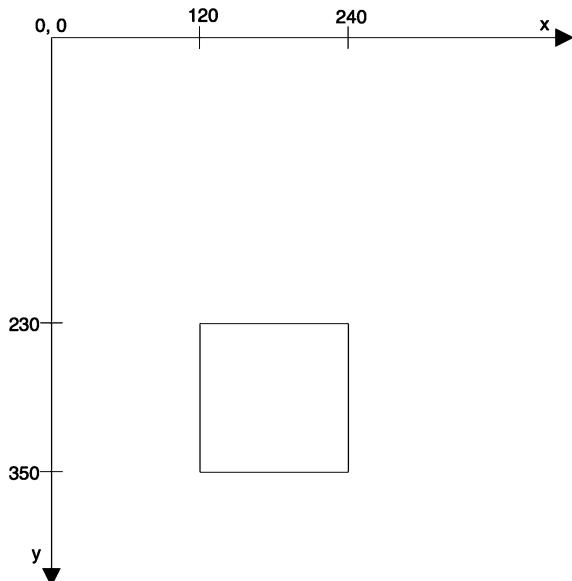
Figure 3.9 shows the position of the rectangle in the Smalltalk coordinate system.

Graphic Representation of Geometrical Objects

At this point, we will make a small detour to show the simple methods used in VisualWorks to draw a geometrical object such as our rectangle in a window. For that purpose, somewhat simplified, one must create a “window object” or, more precisely, an instance of the class `ScheduledWindow`. The object to be depicted must be added as a component to this window object, before finally “opening” the window and displaying the object on the screen. This is accomplished by using the following sequence of Smalltalk expressions:

```
| aScheduledWindow |
aScheduledWindow := ScheduledWindow new.
aScheduledWindow
    component: (Rectangle
        origin: (120 @ 230)
        corner: (240 @ 350))
```

Fig. 3.9 The Smalltalk coordinate system showing a rectangle



```
asVisualComponent.  
aScheduledWindow open.
```

The class method `new` of the class `ScheduledWindow` creates a new instance of this class. One can use the message `component:` to add an object (to be displayed at a later time) as a component to a window. In order to actually be able to draw the rectangle, before presenting it to the window as a component, one must first use the message `asVisualComponent` to transform it into a visual component, that is, into an object that can be drawn on the screen. Finally, the message `open` displays the window on the screen, along with any components it contains. Once the Smalltalk sequence shown above is entered into the workspace and executed with the **Do it** command, SmaViM will in all likelihood open too small a window for the rectangle it contains to be visible. The mouse, though, can easily enlarge the window and show the result seen in Fig. 3.10.

3.2 Literals

Among the basic elements of a programming language one finds objects that can be used simply by writing them into the program. These singular and fully contained objects are called *literals*. Numbers are among the simplest literals in any programming language. From the very beginning of this book, we've made use of our ability to just write down numeric objects such as 5 or -3.14 . (See Chap. 2.) Objects of this type may be said to have existed from the very start of programming; they don't need to be created. Or, one could

Fig. 3.10 Rectangle drawn in
a ScheduledWindow



//

also say that the compiler creates them automatically whenever it encounters a literal. This section examines in somewhat greater detail the objects of classes that are represented as literals.

Smalltalk recognises five types of literals:

1. Numbers
2. Characters
3. Character strings
4. Symbols
5. Arrays

Numbers

We have already encountered numeric literals in two formats:

- As whole numbers—whole numbers are instances of the class `Integer`. Examples: `15`, `-23`, `20345698763512345`
- As so-called floating-point numbers—floating-point numbers are instances of the class `Float`. Examples: `3.14159`, `-2.1828`, `3.34E-10`. In mathematical format the last number would be written $3.3 \cdot 10^{-10}$.

There are other numeric classes, which means there are also other formats for numeric literals. We will discuss these at greater length in Sect. 8.1.2.

Characters

Characters are instances of the class `Character` and represent individual symbols of an alphabet. Character literals are written with a dollar sign immediately preceding the

character to be represented. Thus

```
$g  
$3
```

stand for the letter “g” and the numeral “3”. Please be aware that the numeral is used here as an alphanumeric character, and has nothing to do with the number 3.

The alphabet consists—at least—of:

- The lowercase letters \$a...\$z
- The uppercase letters \$A...\$Z
- The numerals \$0...\$9
- A series of special characters
 - \$+ \$/ \$~ \$< \$> \$= \$@ \$% \$! \$& \$? \$! \$,
 - \$[\$] \${ \$} \$; \$\$ \$# \$: \$. \$.

Stated simply, one can say that alphanumeric characters include all characters that can be entered from a keyboard. That means that one can also use special characters used in other languages, such as those with umlauts (\$ä) or other diacritics (é, à, ñ).

Character Strings

Character strings are instances of the class `String` and represent a series of characters. In this book, especially in Chap. 2, we have used them to represent text intended to appear in dialogue windows. Character string literals are written as a series of characters enclosed between single quotation marks. For example,

```
'this is a character string'  
'345'  
'John''s car'
```

If you need to use a single quotation mark within a character string, you must enter it twice. The third character string therefore consists of the characters

```
$J $o $h $n $' $s $c $a $r
```

Note also that the second character string consists of the individual characters \$3, \$4 and \$5 and has nothing to do with the number 345.

`$a` and ‘`a`’ are two different objects. The first is an instance of the class `Character`, while the second is an instance of the class `String`, which includes as its only content the character `$a`.

Evaluating the expression

```
'a' = $a
```

produces `false`. One can query a character string to see if it includes a particular character by sending it the keyword message `includes:`. Evaluating

```
'a' includes: $a
```

produces `true`.

Symbols

Symbols are instances of the class `Symbol`. Symbols are close relatives of character strings, which SmaViM handles in a special way. Each symbol is stored just once in SmaViM, even if it occurs as a literal in various places. The system uses symbols as names of classes and methods. Because of the special way in which they are used, the system can efficiently check to see if a particular symbol already exists, which can prevent, for example, assigning a class name twice.

Symbol literals are represented by character strings with the character “#” (pound or hash sign or octothorpe) appended at the beginning. If the character string contains characters other than alphanumeric characters, they must be enclosed between single quotation marks. The following are examples of symbols:

```
#January  
#Symbol  
#'this is also a symbol'
```

Character strings and symbols are handled differently because of the general principle of identity, or identity of objects, which will be discussed in detail in Sect. 11.4.

Arrays

Instances of the class `Array` are collections in which are placed a certain number of any type of objects; the specific number is determined when the object is created. That means that a character string might be regarded as a special kind of array, whose included objects must be instances of the class `Character`. Strings and arrays are very similar to one another; both are collection classes or container classes that play a particular role in programming. For that reason, these classes are discussed in detail in Chap. 10.

We discuss arrays in this section because they can be used to form literals. An array literal is introduced by a pound or hash sign, followed by an enumeration of the objects to be contained in the array, enclosed between parentheses. The array

```
#(123 $x 'hello' #Sunday)
```

contains four objects: a number, a character, a character string and a symbol. The elements (or components) of an array can be instances of absolutely any class; for array literals, however, only elements are possible that can in turn themselves be characterised by a literal. Later on we'll learn other methods for creating arrays and for adding any kind of elements to them. Among array components though are arrays themselves; in other words, one can compose array literals that themselves contain array literals. The following array literal contains a number, a character, a character string, and another array, which in turn contains a number and a character:

```
#(4 $4 'four' #(5 $5))
```

Within array literals, one can omit the `#` character, both for nested arrays and for symbols. The following array contains a number, an array and a symbol:

```
(#7 ($7 'seven') seven)
```

Chapter 4 deals with an initial application for arrays.

3.3 Variables and Assignments

Variables serve the purpose of giving names to objects, so that those objects can always be used by those names, that is, especially to be able to send them a message. The program for solving quadratic equations that we encountered in Chap. 2 made repeated use of this.

The following Smalltalk program calculates the square root of 2:

```
| root radicand |
radicand := 2.0.
root := radicand sqrt
```

As we've already explained, Smalltalk first requires programmers to declare temporary variables (in this case, `root` and `radicand`) before they can make use of them. The assignment, expressed by the character string “`:=`”, causes the numerical object 2.0 to be associated with the name `radicand`. You can also say that the assignment *binds* the variable to the object or that the variable *references* the object.

Fig. 3.11 Representation of variables and objects

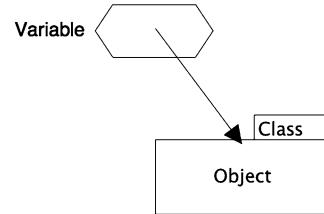
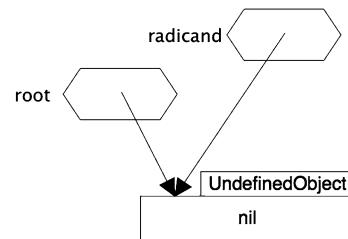


Fig. 3.12 Variables referencing an undefined object



Finally, in the third line, a reference to the numerical object 2.0 occurs. The message `sqrt` is not sent to the variable `radicand`, but instead to the object to which the variable is bound. In addition, in the same line, the assignment binds the result of the calculation (the numerical object 1.41421) to the variable `square root`.

As an illustration, and in order to better understand how the assignment process works, a more technically oriented description now follows. SmaViM must keep track of which variables are valid at any given moment, as well as of which objects exist. This accounting happens as variables and objects are saved in SmaViM. In the following discussion, the schema shown in Fig. 3.11 will be used to describe variables and objects. As the drawing shows, a variable is represented by a hexagon, with the variable name written beside it. An object is shown as a rectangle with the name of the object written within it. In addition, the class of which the object is a member is noted on the upper right edge of the object. The arrow represents the fact that the variable is bound to the object.

Let's look again at the program shown above for calculating $\sqrt{2}$. After SmaViM interprets the first line, it saves the two variables. The variable declaration merely establishes the variable name. Until an assignment explicitly binds a variable to an object, the variable remains bound to the undefined object represented by the pseudo-variable `nil`. After the declaration of the two variables, the status in memory is what is shown in Fig. 3.12. One can see that two different variables can be bound to the same object. The object represented by the pseudo-variable `nil` is the only instance of the class `UndefinedObject`. This exists exactly one time in memory.

In the second line of the program, the number 2.0 is bound to the variable `radicand`. It loses its former binding to the undefined object, resulting in the memory situation shown in Fig. 3.13.

Similarly, the third line of the program results in binding the variable `root` to the result of the solution of the expression `radicand sqrt`, which results finally in the status shown in Fig. 3.14.

Fig. 3.13 Variable radicand bound to a Float object

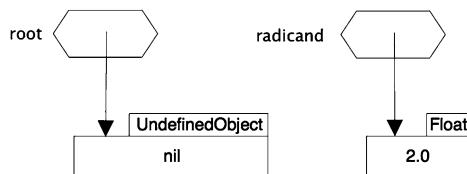
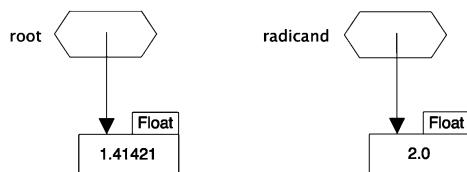


Fig. 3.14 Variable root bound to a Float object



Longevity of Objects

Temporary variables exist in memory only for the duration of the execution of the Smalltalk program. In other words, at the moment that the execution of the lines selected in the workspace begins (with **Do it**, for example), SmaViM writes the variables to memory; when the execution stops, it removes them. This behaviour also has consequences for the objects bound to those variables. The general rule is that an object exists in memory only as long as there is at least one reference to this object. When the final reference finally disappears, for example, when a temporary variable ceases to exist, the life of the object also ends. The memory space occupied by the object is automatically reallocated to free (unoccupied) memory, where it is once again available for the creation of new objects.

Section 3.4 discusses further consequences of the fact that in Smalltalk, variables always contain references to objects.

► **Note:** This automatic garbage collection has been a significant feature of Smalltalk VMs from the very beginning. In the meantime, it has become commonplace in the implementation of other object-oriented languages, such as Java. The alternative to automatic garbage collection is to assign the task to the application programmer, who then becomes responsible for the correct reservation and release of memory space. But decades of experience in software development have shown that these additional tasks ask too much of many programmers. Nevertheless, it cannot be denied that automatic garbage collection is a process that demands a great deal of computer time, which is then not available for carrying out application programs. That in turn places high demands on the hardware's capacity. This fact has certainly contributed to the fact that this concept has only slowly been able to take hold, and that even today there are many software systems that fail to use it out of efficiency concerns.

Variable Scope

One can distinguish between types of variables based on asking in which context they are valid or where they are used. On this basis, one can divide Smalltalk variables roughly into

the following two categories:

- Private variables—accessible for only a single object
- Shared variables—accessible for more than one object

Private variables include instance variables, which, as we have stated, are directly available only using the methods of the object class. Each instance of a class has its own private set of instance variables. Private variables also sometimes appear as so-called *local variables* within methods (see Chaps. 6 and 7). Temporary variables, such as those we use for Smalltalk programs in the workspace, play a similar role. Such variables can be used only within a portion of a program that one characterises as an *unbound method*, because it is not bound to a class the way a “normal” method is.

Shared variables might be so-called *class variables*, which all instances of a class can access. Section 7.3 discusses class variables.

The variables with the greatest scope are the so-called *global variables*, because they can be used everywhere. One example of a global variable is the variable `Transcript`, by means of which a special, always available window of the development environment can be accessed. The system uses this window to write messages for the programmer, and the programmer can in turn create output in the `Transcript` pane (see Sect. 5.3).

► **Note:** Beginning with Version 5 of VisualWorks, it has been possible to define specific scopes for identifiers of variables and classes. These are called *Namespaces* and is a concept that is not found in other development environments. Since the programming examples in this volume are relatively small, Namespaces is not particularly important at this time. Section 7.1.1 contains some basic information on the topic.

Variable Identifier

Variable names must be constructed according to the applicable Smalltalk rules for the structure of so-called *identifiers*. These rules provide that identifiers must consist of any number of alphanumeric characters, but that the first character must be a letter. Some Smalltalk dialects also permit an underscore (“`_`”) as part of an identifier’s name.

A few identifiers have a reserved meaning in Smalltalk and are thus not permitted as variable names:

```
nil, true, false, self, super, thisContext
```

In the world of Smalltalk, a series of conventions impose constraints on identifier names, including the following:

- Identifiers for private variables always begin with a lowercase letter.
- Identifiers for shared variables always begin with an uppercase letter. The same is true for class identifiers, since they also have a global scope.

- Identifiers that consist of multiple words are written in camel case, for example, MonthNames or rectangleSide.

At this point, we'll remind you that when variables are declared in Smalltalk, only the name is stipulated. But no restrictions at all are placed on the types of objects that can be bound to them. One might also say that Smalltalk variables are not *statically typed*. That means that a variable can reference absolutely any object, and that this binding can be changed at any time by reassigning the variable. In the following Smalltalk program, the variable all is bound in turn to instances of the classes Integer, Character and Array:

```
| all |
all:= 25.
all:= f
all:= #('hello' 123 e)
```

3.4 Assignment Semantics

Look at the following Smalltalk sequence:

```
| x y |
x := 2.0 sqrt.
y := x
```

According to the first assignment, the variable x references a Float object and y references the undefined object (see Fig. 3.15). Two possibilities exist for executing the assignment y := x:

- A copy of the object bound to x is made, which is then bound to the variable y. This kind of assignment is referred to as *value semantics*.
- The reference located in variable x is copied into variable y, so that both variables subsequently reference the same object. This kind of assignment is referred to as *reference semantics*.

Smalltalk uses only reference semantics. Figure 3.16 shows the result of the assignment y := x.

If one adds the following assignment to the series of instructions shown above,

Fig. 3.15 Variable x bound to a Float object

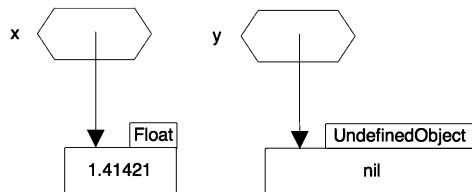


Fig. 3.16 x and y bound to the same object

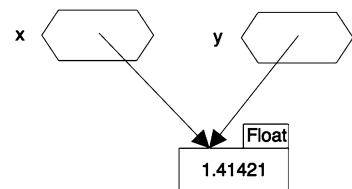
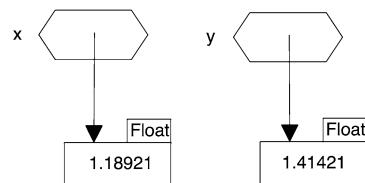


Fig. 3.17 Result of **x := x sqrt**



```
x := x sqrt
```

evaluating the expression creates a new object $\sqrt{2.0} \approx 1.18921$ on the right side of the assignment. The assignment itself then binds variable **x** to this object; **y**'s binding does not change (see Fig. 3.17).

The situation becomes more complicated when there is a change in the status of an object that multiple variables reference. This change can then be seen across all of the variables. The following example illustrates this:

```
| p1 p2 |
p1 := Point x: 100 y: 200.
p2 := p1.
p2 x: 150.
p1 x
```

If one uses **Print it** to execute this sequence in the workspace, the result of the evaluation of the final expression, which displays the **x** coordinate of the **Point** object bound to **p1**, is the value 150. (In this case, the identifier **x** represents a unary message that, when sent to a **Point** object, delivers its **x** coordinate.) How does this come about? After the third

Fig. 3.18 p1 and p2 reference the same point

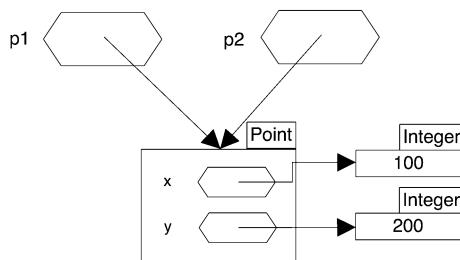
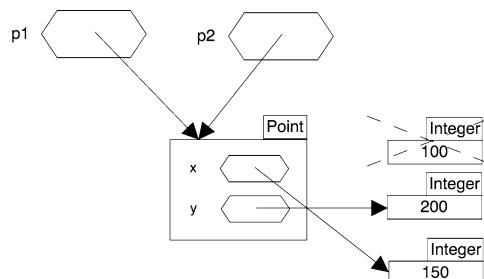


Fig. 3.19 The x coordinate of the point has been changed



line is executed, the memory appears as shown in Fig. 3.18. The two instance variables x and y determine the structure of the Point object, which in this case, reference the two Integer objects 100 and 200.

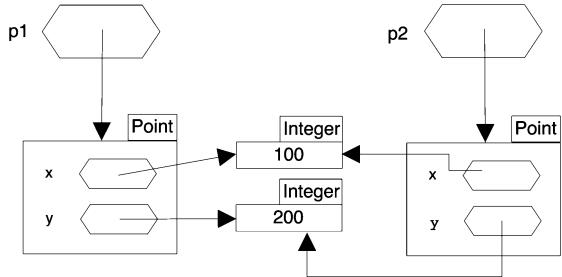
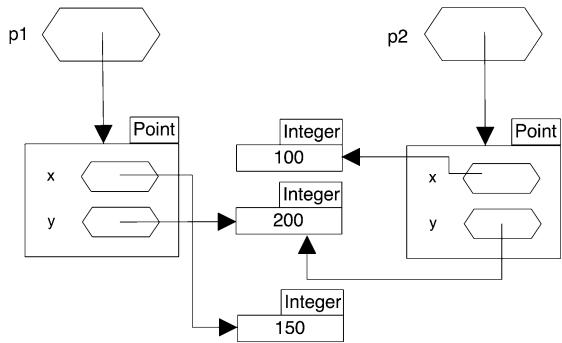
In the fourth line, the keyword message x: 150 is sent to the object bound to variable p2. The object responds to this message by changing its internal status, by binding the instance variable x to the number 150. The expression, though, has no effect on the bindings of either p1 or p2, which results in the situation shown in Fig. 3.19.

It is, however, still true that only a single Point instance exists, to which the two variables are bound, and so the object bound to p1 obviously responds with 150 when it is queried about its x coordinate, as happens in the last line of the program.

The fact that a change in the status of an object—as occurred in the above example—becomes visible over multiple variables, is referred to as a *side effect*. Side effects frequently cause programming errors and can occur when objects are manipulated without bearing in mind that the object is bound to a different variable at another location in the program.

In the above example, if one had wanted two independent points—even though they were initially identical—one could have achieved this by using the following series of instructions:

```
| p1 p2 |
p1 := Point x: 100 y: 200.
p2 := Point x: 100 y: 200.
```

Fig. 3.20 Two equal points**Fig. 3.21** Two different points

This results in the memory structure shown in Fig. 3.20. Any subsequent manipulation of the point bound to p1 has no effect on the point bound to p2. A subsequent execution of the expression `p1 x: 150` yields the memory structure in Fig. 3.21.

3.4.1 Using Object Explorer

Object Explorer is an expansion of the VisualWorks development environment that was originally developed by Kent Beck and is now part of VisualWorks. It becomes available, though, only after one has installed it.⁵ Once the package has been installed, all objects understand the message `explore`.

Object Explorer permits graphic depiction of object structures, such as we have been using in the previous sections (for example, Figs. 3.19 and 3.20). For that reason, it is particularly suited to helping beginners visualise object structures. For example, if one enters the instruction in the workspace `(100@200) explore`, a graphic like the one in Fig. 3.22 appears. Object Explorer uses a box to represent an object. The name of the object's class appears in the first line, followed by a list of the instance variables. Clicking on an instance variable displays any objects that it points to; in our example, the instance variables point to the numbers 100 and 200. Figure 3.23 illustrates how objects are displayed as literals when a literal representation for them exists.

⁵ Appendix A.2 explains how to do this.

Fig. 3.22 Representation of a Point object in Object Explorer

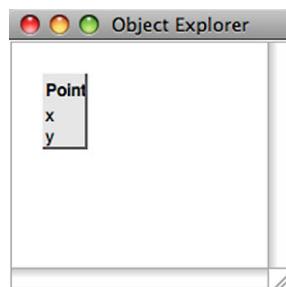


Fig. 3.23 Point object with its bound instance variables

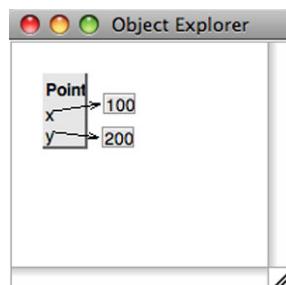
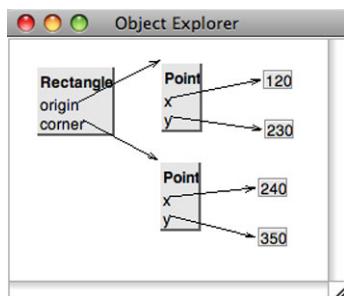


Fig. 3.24 Representation of a Rectangle instance



The Rectangle object shown in Inspector in Fig. 3.8 can be viewed in Object Explorer using the following expression:

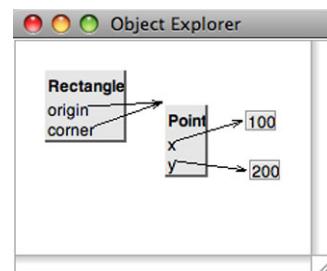
```
(Rectangle origin: 120 @ 230 corner: 240 @ 350) explore
```

Fig. 3.24 shows the result.

And the final example in this chapter illustrates reference semantics. The program code

```
| p |
p := 100@200.
(Rectangle origin: p corner: p) explorer
```

Fig. 3.25 Representation of a meaningless Rectangle instance



initially creates a `Point` object and stores it in variable `p`. This variable is then used to define the two endpoints of the rectangle's diagonals. The Object Explorer display (see Fig. 3.25) shows that only a single `Point` object actually exists, which both instance variables (`origin` and `corner`) reference. Of course, it does not make much sense to construct such a `Rectangle` object.

Chapter 2 introduced algorithm design, with an emphasis on the systematic development of case-by-case distinctions that need to be taken into account when solving a given problem. For the description of an algorithm, this examination yielded two basic elements for the ordering of instructions:

- The *sequence*, or simple sequential execution of instructions
- The *case-by-case distinction*,¹ or the execution of instructions dependent upon the validity of a condition

The algorithm used to solve a quadratic equation (see Sect. 2.3) made extensive use of both elements.

Many problems, though, require yet another basic algorithmic pattern: *repetition*.² Repetitions also occur in the algorithms of daily life. For example, in the days before electric mixers, a cookbook might have contained the following algorithm concerning whipping egg whites:

“Beat the egg whites with a whisk until stiff peaks form.”

If you regard a single “beating of a whisk” as a basic instruction, then you are to repeat that instruction until a particular condition has occurred. You could slightly transform the algorithm:

```
as long as: egg whites have not formed stiff peaks  
repeat: beat the egg whites with the whisk
```

¹Frequently called a *branching*.

²Frequently called a *loop*.

You can see that this kind of repetition structure consists of two parts:

- A condition
- An instruction (or series of instructions) that must be repeated as long as the condition is true.

With a case-by-case distinction, the condition is tested once and if the condition is satisfied, the series of instructions is executed exactly one time. For a repetition, in contrast, as long as the condition is true, it is retested every time the series of instructions has been executed. This process ends when the condition is no longer true. At that point, the algorithm continues with the next instruction following the repetition structure.

The next section looks at a typical data-processing problem—the solution algorithm, which requires a repetition structure. In the process, we'll also describe again the process of step-by-step refinement in the design of algorithms.

4.1 Searching for a Maximum Value

The greatest number in a series of whole numbers is to be determined. First we'll refine the statement of the problem.

Problem:

Given: A finitely not empty set of *count* whole numbers

Sought: The maximum value of this set, that is, the greatest number

Although *count* is finite, it can be of any size, and so we can best visualise it using the following example.

Example:

To be able to grasp what we're doing, let's select a set with 10 elements, that is *count* = 10. Let the set be called *numbers*. Now let

$$\text{numbers} = \{3, 2, 17, -9, 81, 14, 5, 23, 8, -12\}$$

If we call the maximum value *greatestNumber*, then the following is true:

$$\text{greatestNumber} = 81$$

Although the result is true, it doesn't reveal to us how we got there. An algorithm for an automatic solution to the problem requires

- The statement of a procedure and
- A formulation of the statement that can be processed by a machine, that is, a written program.

We start by positing a procedure, which we will subsequently convert into a series of formulations, each one better suited to be carried out by a machine. This is the process known as *step-by-step refinement* (Wirth 1971).

When we are looking for a process, that is, when we want to find a general principle for solving a problem, we first have to precisely describe the problem we want to solve. Such a description is called the *specification* of the problem to be solved automatically. Using mathematical notation, we write:

- I*: The algorithm requires as input the non-empty set $\text{numbers} \subseteq \mathbf{Z}$
 \mathbf{Z} stands for the set of whole numbers.
- O*: The algorithm yields as its output the maximum value of numbers , that is
 $\text{greatestNumber} = \text{maximum}(\text{numbers})$

The solution to the problem that we want our machine to solve is formulated precisely enough when we have described the input-output behaviour of the automaton that we want to solve the problem according to the requirements of the algorithm. For that reason, we speak of the specification of the input-output relation or, in short, the I/O relation.

This relation describes the input-output behaviour of an algorithm in the following manner:

If the input precondition *I* has been fulfilled before the execution of an algorithm, then the output confirmation *O* applies after the execution.

The specification of the I/O relation for the problem of finding the greatest number requires the precise description of the function *maximum*. Let *M* be a non-empty set of whole numbers. Then the greatest number, *maximum* (*M*), is the greatest number in the set *M*; it can be described by the following property:

$$\text{maximum}(M) \in M, \quad \forall x(x \in M \rightarrow x \leq \text{maximum}(M))$$

This gives us a precise mathematical specification of the problem we want to solve. In general, this represents a good basis for program development, which is unfortunately only rarely found in software development.

If the machine that is to solve the problem is SmaViM, then our first step in developing a solution could be to reformulate the mathematical specification in Smalltalk terms. We can use an array to represent a set of numbers. To specify the sample quantity shown above, in Smalltalk we would write:

```
| numbers |
numbers := #(3 2 17 -9 81 14 5 23 8 -12)
```

And if an array object were capable of understanding the message `maximum`, then the solution to the problem would be:

```
| numbers |
numbers := #(3 2 17 -9 81 14 5 23 8 -12).
numbers maximum
```

But there is no method suited to this message. Since arrays in Smalltalk can contain not just numbers but any kind of object, such a method would work only if a greatest number were mathematically defined for the elements of the array.

That is, we have to refine the algorithm, for which we choose the following.

Method:

It is clear that each element of the array must be examined, because each one could be the greatest number.

First of all, we make use of the fact that the greatest number in a set must be one of the elements of the set, and so we initially and arbitrarily assume that the first element of the set is the greatest number. We note that this element is—up until now—the largest element (in comparison to any other elements of the set that we have already examined) and call it *intres*.

Next, we examine in turn the other elements of the set and test whether the element we are currently examining is greater than the previous greatest number (*intres*) that we noted earlier. At this point, two cases can occur:

- If the new element is larger, we note its value and again call it *intres*, thereby forgetting the element that we had previously noted.
- If it is not larger, then we have nothing else to do with it.

Next, we proceed to examine the next element.

We repeat this process as long as there are elements in the set *numbers* that have not yet been tested. After that, we make *greatestNumber* equal to *intres*.

It must be clear to most readers at this point that the algorithm we described above in everyday language is sufficiently plausible, but before we formulate a program, we shall prove that this algorithm represents a correct solution in the search for the problem of searching for the greatest number.

According to our initial assumption, the set *numbers* contains at least one element. We'll additionally assume that we number the elements of the set consecutively. Then we can conduct the following inductive proof:

Claim: After executing the algorithm, intres represents the greatest number of the count elements in the set numbers .

Basis: Let $n = 1$ and intres be the first element in numbers . In that case, intres is the largest of the first n ($n = 1$) elements in the set numbers .

If $\text{count} = 1$, then nothing more needs to be shown. Otherwise, we argue the following:

Inductive hypothesis: Let $\text{count} > 1$, $n < \text{count}$ and intres the largest of the first n elements in the set numbers .

Inductive step: As described in the algorithm, we test the $n + 1$ -th element in the set numbers and compare it to intres . If it is larger, then we set intres equal to this $n + 1$ -th element. In that case, intres is the largest of the first $n + 1$ elements in the set numbers .

Thus we have shown that, for $n = \text{count}$, the following is true:

$$\text{intres} = \text{maximum}(\text{numbers})$$

If we apply the algorithm to a finite, non-empty set of whole numbers, it yields the greatest number of this set of numbers, if the algorithm ends.

The algorithm requires that the execution is to stop once the last element (the one numbered count) has been examined. We say that the algorithm breaks, or that it *terminates*. That way we can record the fact that the algorithm is correct with regard to the specified I/O relation.

Now we consider the problem of a formulation of our procedure suited for running on a computer. The procedure tells us that, at the start, the first element in the set is to be selected as the provisional greatest number (intres), since it was given that numbers is non-empty.

This instruction represents a so-called *initialisation* for the subsequent repetition.

After the initialisation, there follows a series of instructions, the execution of which is repeated as long as the specified condition applies, that is, as long as all elements of the set have not yet been examined.

The instruction that describes this command is called a *repetition* or a *loop*.

We now have the first approximation of a computer-suited formulation:

```

select the first element from the set and call it intres.
as long as all elements of numbers have not yet been
tested repeat
    take the next element of the set numbers and
    call it element.
    if element > intres
        then let element be the greatest number of those
              tested and call it intres.
greatestNumber = intres.

```

This format for writing out the algorithm is referred to as stylised text, which means that it consists of normal text, using however structural elements such as “if...then” or “repeat...until”, and in which text indentations are significant (see below).

The way we have formulated the algorithm up until now raises some thoughts. Since we are ultimately aiming for a computer-readable formulation of the algorithm we are describing, we have to establish which instructions our SmaViM understands and how these instructions are to be formulated.

The first thing we notice is that our algorithm contains a repetition schema:

```
as long as condition repeat {instruction}
```

If the condition has been fulfilled before the first execution of the repetition, then the subsequent instruction is executed. Then the condition is tested again. If it is again fulfilled, the subsequent instruction is executed another time, etc.

The execution of the repetition terminates at precisely the point where for the first time the preliminary condition is not fulfilled.

If—as in our example—several instructions are to be repeated, then we indent them in the lines that follow the `repeat`. That means that the next line that is not indented contains the first instruction that is not part of the loop. In our example, it is the last line.

Each execution of the instruction following the condition is called an *iteration*. When we leave a loop after a finite number of n iterations, we say that the loop terminates after n iterations. If the loop condition is not met the first time the loop is executed, then we say that the loop terminates after 0 iterations. We require that SmaViM be able to execute loops correctly. For that reason, the following model is available in Smalltalk:

```
[ <condition> ] whileTrue: [ <series of instructions> ]
```

In the first block, we write the condition, the validity of which is intended to affect the execution of the series of instructions written in the second block. The keyword message `whileTrue:` thus demands a block both as a receiver and also as an argument. After the series of instructions has been executed, the condition is evaluated another time. If it is fulfilled, then the series of instructions is executed another time. If it is not fulfilled, then the execution of the `whileTrue:` message is terminated.

Now let's look at the condition

```
all elements of numbers have not yet been tested
```

Since the set `numbers` is finite, it is enough that we continuously count the elements that have already been examined and then compare the counter to the preset number of elements in the set `numbers`. If we call such a counter `counter`, then we can replace the quoted condition with the equivalent

```
counter <= count
```

The algorithm is based on a fixed count of elements in the set. If, as we have already done, we use an array to represent the number set, then Smalltalk provides a simple way to determine the number of elements in an array. That is because an array object understands the unary message `size`, so that it can write

```
count := numbers size
```

to determine the number of elements.

Now, the algorithm requires us to access each of the elements individually and in order. We can make use of the fact that in Smalltalk the elements of an array are continuously numbered—beginning at 1—and that we can access them by specifying their number, which is also called their *subscript*. We do this by using the keyword message `at:`, which is provided as an argument the subscript of the element that we want to access.

The instruction

```
element := numbers at: 1
```

assigns the first element of the array `numbers` to the variable `element`.

And now, as a second refinement of the algorithm, we can submit the following Smalltalk-ready formulation:

```
| numbers element counter count intres greatestNumber |
numbers := #(3 2 17 -9 81 14 5 23 8 -12).
count := numbers size.
counter := 1.
intres := numbers at: counter.
counter := counter + 1.
[counter <= count]
    whileTrue:
        [element := numbers at: counter.
         (element > intres)
             ifTrue: [intres := element].
         counter := counter + 1].
greatestNumber := intres.
```

Executing this algorithm in the workspace using **Print it** yields the result shown in Fig. 4.1.

Fig. 4.1 Determining the greatest number of a number array

The screenshot shows a 'Workspace' window with a menu bar: Page, Edit, Smalltalk, Options, Help. The 'Page' tab is selected. Below the menu is a toolbar with 'Page 1' and 'Variables'. The main area contains the following Smalltalk code:

```

| numbers element counter count intres greatestNumber |
numbers := #(3 2 17 -9 81 14 5 23 8 -12).
count := numbers size.
counter := 1.
intres := numbers at: counter.
counter := counter + 1.
[counter <= count]
whileTrue:
[element := numbers at: counter.
(element > intres)
ifTrue: [intres := element].
counter := counter + 1].
greatestNumber := intres.
greatestNumber 81

```

The variable 'greatestNumber' is highlighted with a blue selection bar, showing its value as 81. At the bottom of the workspace window, there are buttons for 'Text or Smalltalk code' and 'All'.

Moreover, the Smalltalk program shows clearly that assignments are nothing at all like equations in a mathematical sense. Otherwise the instruction

```
counter := counter + 1
```

would be meaningless. No value for `counter` could satisfy such an “equation”. If, on the other hand, one interprets this instruction correctly as an assignment, then the meaning is immediately clear. For example, if `counter` has the value 6 before the execution, then evaluating the expression

```
counter + 1
```

yields the value 7.

This value is then assigned to the variable `counter`. The assignment has the effect of replacing the old value of `counter` (6) with the newly calculated value (7).

► **Note:** A more precise formulation of the effect of assignments would be: Before execution, the variable `counter` is bound to the object 6; the assignment now binds it to the object 7. The same variable is used to name in succession two different objects.

The solution process described above assumes that the number set is not empty, that is, that the array in the Smalltalk program contains at least one element. When the array is presented as a literal, that is of course easy to accomplish. Imagine, though, what would happen if the determination of the maximum value were to be used in a context in which the content of the array was to be calculated. In that case, one would not necessarily want

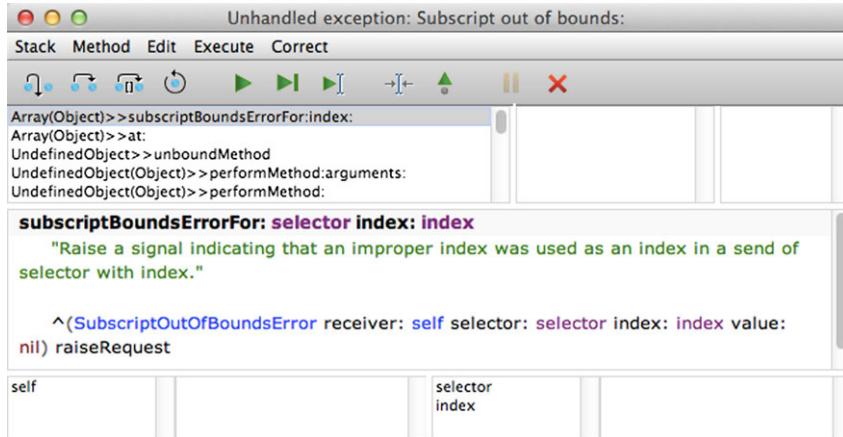


Fig. 4.2 Starting window of the debugger

to rely on the required property of the array, that it contain at least one element. Before we discuss a way of handling this problem, let's see how SmaViM reacts when we present it with an empty array, but leave the rest of the program unchanged. In that case, the program would look like this:

```
| numbers element counter count intres greatestNumber |
numbers := #( ).  
count := numbers size.  
counter := 1.  
intres := numbers at: counter.  
counter := counter + 1.  
[counter <= count]  
    whileTrue:  
        [element := numbers at: counter.  
         (element > intres)  
             ifTrue: [intres := element].  
         counter := counter + 1].  
greatestNumber := intres.
```

If you try to run this program in the workspace, you will get the error message shown in Fig. 4.2. The error “subscript out of bounds” means that an attempt was made to access an array using a subscript for which there was no array element.

It is easy to see that the error occurs in the fifth line of the program, where, in the expression

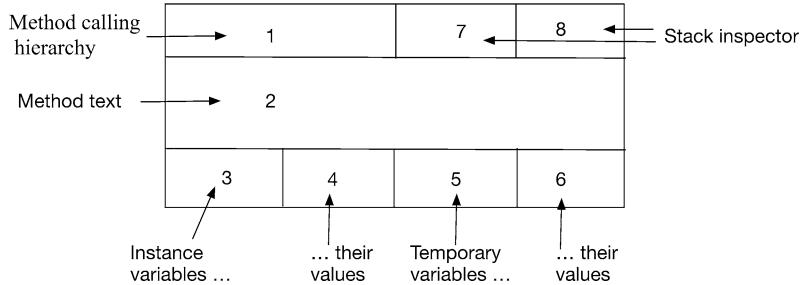


Fig. 4.3 Schematic diagram of the debugging window

```
numbers at: counter
```

the variable `counter` has the value 1, which is used as a subscript to access the array `numbers`, which, however, contains no elements.

One can also use the VisualWorks development environment tool that displays the error message, the so-called debugger. This tool provides several possibilities for finding errors.

At this point, we will not attempt to describe the variety of ways that the debugger can be used, but only to show how one can find the place in the program that caused the exception. Figure 4.3 shows a schematic representation of the structure of the debugging window.

Field 1 displays the so-called context stack of the methods that were active at the point where the error occurred. This is where all message sends appear that were still waiting for their response value when the program aborted. The top line contains the message that was the immediate reason that the debugger window opened, in this case the message `subscriptBoundsErrorFor: index:..`. As a rule, this line is not very interesting for finding errors, because it doesn't say very much about the reason for the error. The message selectors or method names are to the right of the double greater-than sign (`>>`); the left of the sign is the class of the receiver of this message.

If we continue to look down the list, we encounter a message that is part of a method that we ourselves wrote, and which probably led to the error. A piece of a Smalltalk program that was executed in the workspace is not a “real” method, because it does not belong to a class. Such workspace programs receive a temporary replacement name `unboundMethod` and are assigned to the class `UndefinedObject`. This method is found in the third line.

For the most part, the lines after that do not need to trouble the programmer. They are connected with the calling mechanism for methods, which is in fact itself programmed in Smalltalk, but is in all probability not responsible for the failure that occurred.

If one selects this method—as shown in Fig. 4.4—two things happen. On one hand, the method text is displayed in Field 2 (see Fig. 4.3). In this case, that is the series of instructions from the workspace, and the message that led to the error is highlighted. In

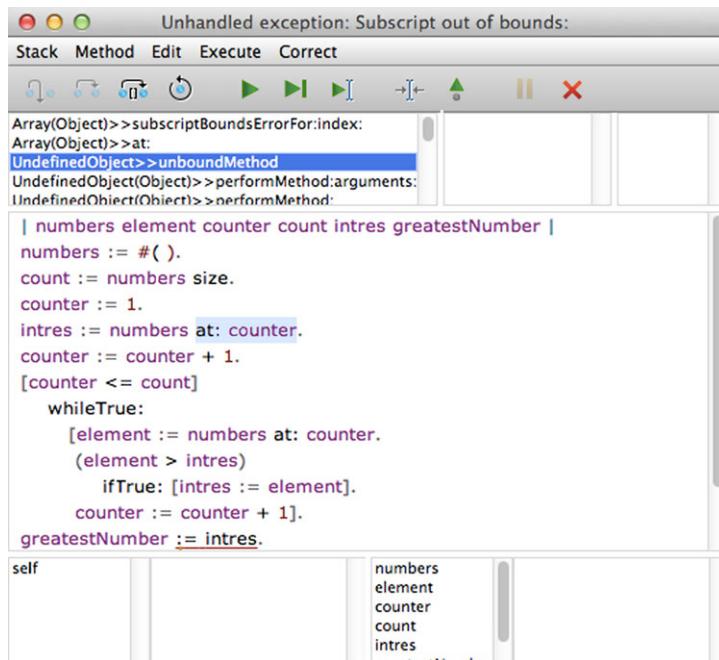


Fig. 4.4 Display of method text and variable values in the debugger

addition, in the lower part of the debugging window, in Field 5, all temporary variables are listed, the values of which can be displayed in Field 6 by clicking. Figure 4.4 shows the value for the variable `counter`.

This lets you find the location of the error very quickly. Chapter 9 discusses other applications for the debugger.

If you don't want your neglect to let the program get into an error situation by encountering an empty array, one could also expand the algorithm by including the test

```
count > 0
```

But there's really no point in continuing the program if that condition is not fulfilled. Nevertheless, it's also possible to abort the program with a meaningful error message, by making the following changes:

```
| numbers element counter count intres greatestNumber |
numbers := \#( ).
count := numbers size.
```

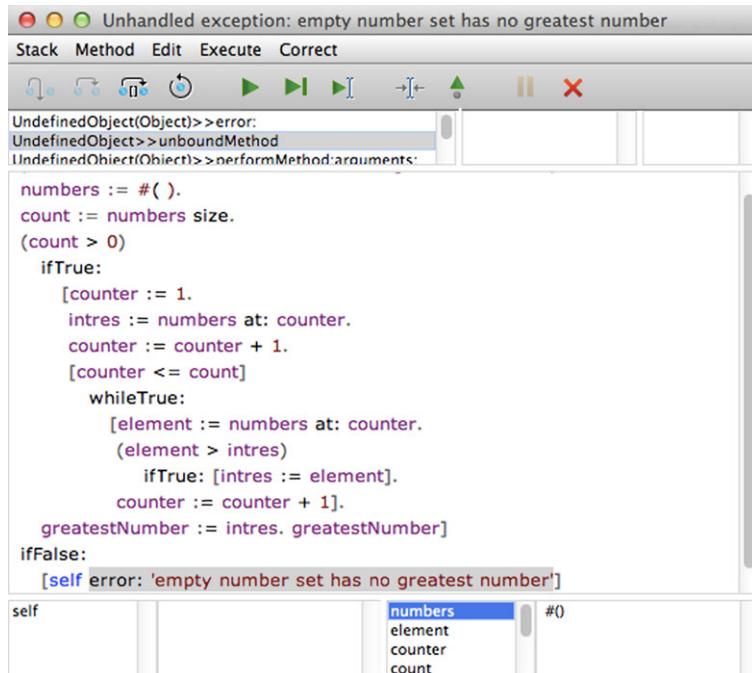


Fig. 4.5 Debugger window with its own error message

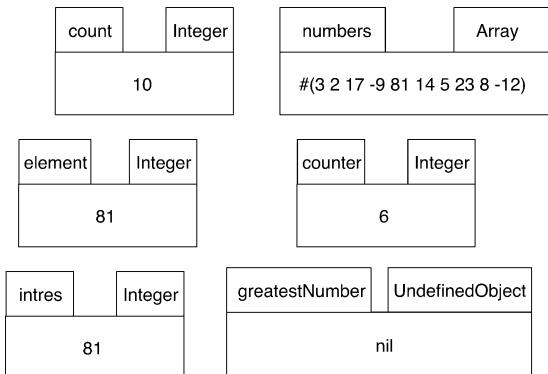
```

(count > 0)
ifTrue:
[counter := 1.
intres := numbers at: counter.
counter := counter + 1.
[counter <= count]
whileTrue:
[element := numbers at: counter.
(element > intres)
ifTrue: [intres := element].
counter := counter + 1].
greatestNumber := intres.]
ifFalse:
[self error:
'empty number set has no greatest number']

```

If you run the program in the workspace, the debugger window shown in Fig. 4.5 will be displayed. It is called by the message `error:`, which expects a string as an argument and

Fig. 4.6 Memory contents after four loop iterations of the program greatest-number search



which is understood by every object. In this case, we are using the pseudo-variable `self`, which is always available and which represents the object that is the receiver of the message, whose method is in the process of being executed. (See also Sects. 6.2 and 11.2.2.)

When you test algorithms or their associated programs, it's important to examine certain specific cases and to test the proper functioning of the program when it encounters them. In the search for the greatest number, these cases would be

- The number set is empty—the situation we just tested
- The number set contains exactly one element. Think for a minute about how the program operates: The series of instructions in the loop (the argument block for the `whileTrue:` message) will not be executed at all.

In Sect. 3.3, we already described how the variables and their values, that is, the objects to which they point, are stored in the SmaViM storage. Think of the set of variables used in a program as its memory. The content of the memory at a particular point in time reflects the status of the program's processing at that very point. The assignments in the program are continually changing this processing status.

For example, Fig. 4.6³ shows the state of the memory for our greatest-number search executed against the number set

```
#(3 2 17 -9 81 14 5 23 8 -12)
```

after the fourth iteration of the loop, that is, directly after the instruction

³In order to save space, the variables in this drawing are not shown as hexagons, as they were, for example, in Fig. 3.11, but are written directly on the objects.

```
counter := counter + 1
```

The value of `greatestNumber` is (still) undefined.

A snapshot of the execution of a program that notes down the variables along with their values at a particular time is part of the memory status of a machine. It is called the *state of execution*. The sequence of all states of execution from the beginning of the execution of a program is called the *execution sequence* of the algorithm in question.

In this context, when we speak of *executions*, *states of execution* and *execution sequences* of programs, we mean the executions, states of execution and execution sequences of the algorithms described by their various programs. In Sect. 9.5, we will learn how the debugger can be used to run step-by-step through the individual states of execution and to review the relevant memory status.

Summary

In this chapter, we introduced the repetition, or loop, as the third algorithmic pattern. We got to know it in the format

as long as condition repeat instruction

or in its Smalltalk version

```
[ <condition> ] whileTrue: [ <series of instructions> ]
```

This form is generally valid, because it can be used to formulate all algorithms that require a repetitive structure. That means that the `whileTrue:` message is also sufficient for Smalltalk programming. But because there are several other special repetitive patterns, Smalltalk offers several additional messages. A few of them will be discussed in the next section. These messages allow you to formulate programs that are shorter, more pointed and frequently also more legible. In contrast to the `whileTrue:` message, though, they are not always usable.

► **Note for people who know conventional higher programming languages like PASCAL:** Conventional programming languages also have various language elements for formulating loops. As a rule, they have a so-called `while` instruction that corresponds in meaning to the Smalltalk `whileTrue:` message. In addition, there are usually additional loop instructions that are, however, not necessary from an algorithmic point of view. These include, for example, the `for` instruction for the so-called counting loop (see below).

In the next section, we will see how the algorithm for the greatest-number search can be formulated to be substantially shorter. From the viewpoint of an experienced Smalltalk programmer, the formulation we have selected in this section is simply impossible. We apologise for this at this point, noting that the concentration on the algorithm in this discussion required initially the introduction of a generally applicable loop structure.

4.2 Additional Smalltalk Messages for Loops

4.2.1 Count Loops

There's another kind of iteration that we call *count loops*, which can be used whenever the programmer knows the number of times the loop should run, that is, when the number is fixed before the loop starts. In the algorithm described in the last section for finding the greatest number, the loop runs $(\text{count} - 1)$ times, because—as long as the set numbers contains more than one element—the loop processes the 2nd, 3rd, ..., countth element.

In Smalltalk, you can use the `timesRepeat:` message to express the n -times repeat of a series of instructions:

```
n timesRepeat: [ <series of instructions> ]
```

The recipient of this message is a whole number n . When n receives the message, it reacts by evaluating the block transmitted as an argument. That lets us rewrite the program for finding the greatest number as follows:

```
1 | numbers element counter count intres greatestNumber |
2 numbers := #().
3 count := numbers size.
4 (count > 0)
5 ifTrue:
6   [counter := 1.
7    intres := numbers at: counter.
8    counter := counter + 1.
9    count - 1
10   timesRepeat:
11     [element := numbers at: counter.
12      (element > intres)
13        ifTrue: [intres := element].
14      counter := counter + 1].
15   greatestNumber := intres. greatestNumber]
16 ifFalse:
17   [self error:
18    'empty number set has no greatest number']
```

The program has changed in the ninth and tenth lines only, but has not gotten any shorter. There's really no advantage to using the `timesRepeat:` message here. There's really an advantage only if exactly the same procedure must be repeated n times. That's not the case in the example, where the value of `counter` changes from one loop run to the

next, and a new element of the `numbers` set is accessed for each run. In other words, the `timesRepeat:` message is useful only for certain specific cases.

4.2.2 Interval Run

As was true for count loops, *interval runs* presuppose that the programmer knows the number of times the loop should run, that is, that the number is fixed before the loop starts.

One of the properties of the program for finding the greatest number is that it runs for increasing values of the variable `counter`, beginning with the value 2. In the loop, the value must always be incremented by 1. This is a common situation, which can be mastered somewhat more easily by using the message `to:do:.` (In a number of common programming languages, the `for` command accomplishes the same thing.) The message is used like this:

```
n to: m do: [ :i | <series of instructions> ]
```

The identifiers `n` and `m` represent two whole numbers that form the limits of a closed interval that increments by 1. The keyword `do:` causes the block transmitted as an argument to be run once for every element of the defined interval. When that happens, the *block variable* `i` takes on the current value of the interval each time the loop runs.

This teaches us about a new syntactic element of the Smalltalk language: the block variable. Within a block, block variables are declared after the opening square bracket; each variable in the declaration is identified with a preceding colon. A block variable is valid only for the series of instructions appearing after the pipe (`|`). It can be used only in that case, at which point the colon is not set.

As an example, the following program can be used to calculate the sum of whole numbers from 1 to 100:

```
| sum |
sum := 0.
1 to: 100 do: [ :i | sum := sum + i].
sum
```

When you use **Print it** to execute the program in the workspace, you get the result 5050. Within the block, the block variable `i` sequentially takes on the values 1, 2, 3, ..., 100. Note that you cannot assign values to a block variable.

Now let's apply the interval run to our greatest-number-search program:

```
| numbers element count intres greatestNumber |
numbers := #(3 2 17 -9 81 14 5 23 8 -12).
```

```

count := numbers size.
(count > 0)
  ifTrue:
    [intres := numbers at: 1.
     2 to: count do: [:counter |
       element := numbers at: counter.
       (element > intres)
         ifTrue: [intres := element]].
     greatestNumber := intres. greatestNumber]
  ifFalse:
    [self error:
      'empty number set has no greatest number']

```

This makes things much more compact. Now we only need the variable `counter` as a *block variable*. This eliminates the initialisation of the variable before starting the loop, and within the loop we no longer need to worry about incrementing the value of `counter`. The `to:do:` message takes care of that when it sequentially assigns it the values of the interval from 2 to `count`.

A variable that can be said to automatically run through the values of a specified interval is also called a *control variable*.

In place of the values 2 and the variable `count` used in our example to establish the interval, one generally uses expressions that provide numeric values that are not necessarily whole numbers. The run variable may not appear in these expressions, because it is valid only within the body of the loop defined by the parameter block. The value of the expressions are not recalculated each time the loop is run. Their evaluation occurs one time only, before the loop is actually executed. You can see that changing the values of these expressions by assigning values within the loop body to variables in the expression has no effect on the number of times the loop runs.

4.2.3 Collection Run

Another loop design that occurs often in Smalltalk is based on the fact that all collection objects—such as, for example, instances of the class `Array`—understand the message `do:.` For instance, you can use it this way:

```
anArray do: [ :e | <series of instructions> ]
```

The result of this instruction is that the series of instructions provided in the parameter block is run once for each element in the collection `anArray`. As this occurs, the current element of the collection is assigned to the block variable `e`.

Using this technique leads to the following variation in the greatest-number-search program:

```

| numbers count intres greatestNumber |
numbers := #(3 2 17 -9 81 14 5 23 8 -12).
count := numbers size.
(count > 0)
    ifTrue:
        [intres := numbers at: 1.
         numbers do: [:element |
             (element > intres)
                 ifTrue: [intres := element]].
         greatestNumber := intres. greatestNumber]
    ifFalse:
        [self error:
            'empty number set has no greatest number']

```

The `do:` message causes the block variable `element` to assume one-by-one the elements of the array `numbers`. Note that the first time the loop runs, the first element stored in `intres` is compared to itself. While unnecessary, it does not impair the functioning of the program.

Summary

This section considered a few variations in which loops were designed using Smalltalk messages:

- The `timesRepeat:` message is used to repeat n times a series of instructions given in a block with no parameters.
- The `to:do:` message permits repetition of a block using a block parameter, which is replaced during each run of the loop with the current value of the interval.
- The `do:` message causes the program to run through all components of a collection (an array, for example, with each component replacing the block parameter).

Smalltalk's class library contains many additional messages that can be used especially for processing collections. The most important are considered in Chap. 10, which deals with collection classes.

Back in Sect. 2.2, it was pointed out that Smalltalk programs have always been developed using a development environment. Such a development environment consists of the following components:

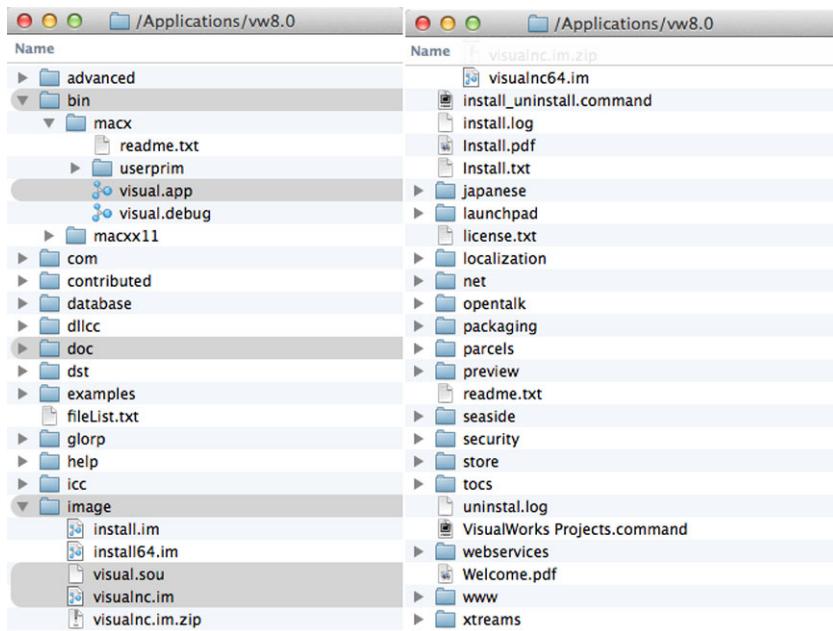
- A graphical user interface that provides the programmer with various tools necessary for program development. These include *Workspace*, *Inspector* and *Debugger*.
- A VM (which we've been calling SmaViM) that's responsible for executing the Smalltalk programs.
- Object storage, which contains all objects that the VM processes, including classes and their methods. This object storage is called the *Image*.

Smalltalk programs reside in object storage in byte code created by the compiler. The compiler is started when one adds a new method to a class or starts to execute a piece of program in the workspace (the so-called `unbound` method).

The next sections provide a glimpse of the VisualWorks development environment. Many of the aspects can be used with appropriate modifications in other systems. But if you want to work seriously with Smalltalk programming, there's no way to avoid studying the manufacturer's original documentation. The information in this chapter should be enough for now, though, to understand the examples in this book and to perform some practical exercises with VisualWorks.

5.1 Overview

As we already said in Sect. 2.2, a VM is a program that must be able to run on the specific hardware on which you want to deploy the development environment. It is thus the only



Name
visualnc64.im
install_uninstall.command
install.log
Install.pdf
Install.txt
japanese
launchpad
license.txt
localization
net
opentalk
packaging
parcels
preview
readme.txt
seaside
security
store
tocs
uninstal.log
VisualWorks Projects.command
webservices
Welcome.pdf
www
xstreams

Fig. 5.1 Directory structure for the VisualWorks installation

platform-dependent component in the development environment. The VisualWorks VM exists in variations for multiple platforms.¹

After installation, the machine where VisualWorks is installed should display a directory structure at least somewhat similar to the one shown in Fig. 5.1. In the course of the installation process, the user can choose any name for the root directory for a VisualWorks installation (vw8.0 in this case).

One needs four files to run a VisualWorks system:

1. An Image file (in this case, `visualnc.im` in the `image` directory) contains the object storage, the Image.
2. `visual.sou` (in the subdirectory `image`) contains the Smalltalk source code of all classes and methods that were delivered in the original Image file in their Smalltalk source code.
3. A file with the same name as the Image file but with the extension `.cha` contains a log of all changes that the programmer has made to the Image. Above all, this file contains the source code for all classes and methods added by the programmer. There is no `visualnc.cha` file in the `image` subdirectory, though, because the Image is still in its original state after installation. Once the `.cha` file has been created, it must always be stored and moved along with the Image file. These two files may never be separated.

¹Among others, Apple Mac OS, Microsoft Windows, Linux.

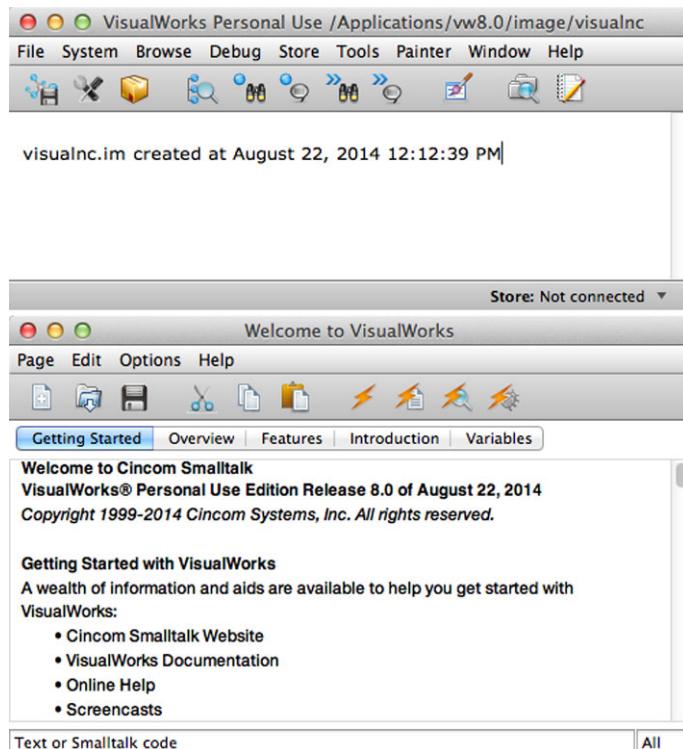


Fig. 5.2 VisualWorks startup screen

4. The file that contains the executable program for the VM is in the .bin subdirectory. Its name depends on the platform, for example:
 - visual.exe for Microsoft Windows
 - visual.app for Mac OS X

An additional important subdirectory is called doc, and contains all of the VisualWorks documentation in PDF format.

5.2 Starting the Development Environment

The `install.pdf` file contains the *Cincom Smalltalk Installation Guide*, in which the start-up procedure for Smalltalk on various platforms is described in the section *Starting VisualWorks the First Time*. In both Microsoft Windows and Mac OS X, the easiest way to start VisualWorks is by double clicking the Image file.

Once you have started VisualWorks, the screen shows two windows (see Fig. 5.2):

1. The window called “VisualWorks NonCommercial” shows the so-called *Launcher*, the control centre for VisualWorks. The next section describes it in greater detail.
2. The window called “Welcome to VisualWorks” is a workspace that contains several tabs with useful tips for handling the development environment and Smalltalk.

This book does not say anything further about this workspace. Nevertheless, it contains much interesting information about Smalltalk and VisualWorks and is very readable. In order to save screen space, though, one can close the workspace. Its contents can be recalled any time via the Launcher’s **Help** menu.

5.3 Launcher with Transcript

The VisualWorks Launcher window (shown in the upper part of Fig. 5.2) contains a menu bar that offers the programmer basic functions of the development environment. Among these are:

- Saving one’s own work by storing the image
- Setting configuration parameters
- Starting tools, for example, the workspace
- Calling online help

Some of the functions accessible from the menus can also be launched via the icons beneath the menu bar. Closing this window also ends your work with the development environment.

Inextricably linked with the Launcher is the so-called *Transcript*, which takes up the lower portion of the Launcher window. The Transcript serves to display system messages, but can also be used by the programmer for output. But before we discuss the Transcript in greater detail, we will provide a few more tips for using VisualWorks.

5.3.1 Creating One’s Own Image

A Smalltalk programmer’s work consists substantially of enriching the original Image with new objects (classes and methods). This work is preserved when, from time to time, a new version of the Image is written to the hard drive. Because one should never change the original Image, in order to be able to access it in case of emergencies, it is advisable before anything else to save the Image under a new name and in a new working directory (if possible), which can be located anywhere.

In order to do this, use the menu item **Save as ...** in the **File** menu of the launcher. In the dialogue window that appears, enter the name you selected for the new Image (for example, `myImage`) and select an appropriate directory. The name of the Image you create in this way appears in the window title of the launcher, and appears automatically as the

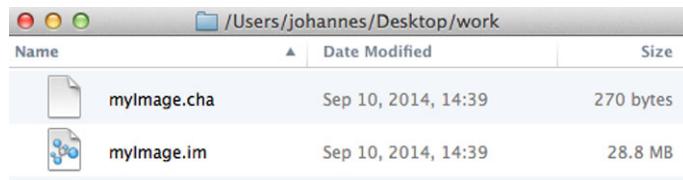


Fig. 5.3 A working directory

file name whenever you try to save your work. This also happens when you try to leave VisualWorks via the menu item **File → Exit VisualWorks ...**. Finally, when you want to start working again on your own Image, one way to start the VM is by double clicking the Image file (`myImage.im` in our example).

Figure 5.3 shows an example of a working directory (called `work` here). The figure shows that the directory contains both the image directory and a `.cha` file, where all changes to the `myImage` Image will automatically be stored.

5.3.2 Setting System Parameters

Once VisualWorks has been installed, you can usually start to use it without any additional configuration. Nevertheless, we recommend that you change a few settings. Change the most important basic settings—those that affect the way VisualWorks functions or the way that it looks—via the *Settings* tool, which you access from **System → Settings** on the Launcher menu. Figure 5.4 shows how the tool looks.

The VisualWorks Home Directory

First of all, you should check to be sure that the root directory—the directory where VisualWorks was installed—is correctly set. Use the **System** screen of the **Settings** tool to check this (see Fig. 5.5). On the right, you see an input field where the full path to the root directory must be entered (in this case, `...vw7.1pu1`). You only need to re-enter this path if you transferred the entire VisualWorks installation to a new storage location. In this case, though, VisualWorks is not entirely platform-independent. On UNIX systems, it is advisable to define a VisualWorks environment variable that contains the root-directory path; otherwise, you have to enter the path each time you begin a new VisualWorks session.

Unless the root-directory path is set correctly, VisualWorks will be unable to find certain files that it needs to run (for example, the `visual.sou` file, described above). If you made any changes here, it is advisable to save the Image (using **File → Save** from the Launcher).

Configuring the Workspace Tool

You can make a number of settings that affect how VisualWorks works from the **File → Settings** menu, but many of these are best reserved for experienced users and shouldn't be

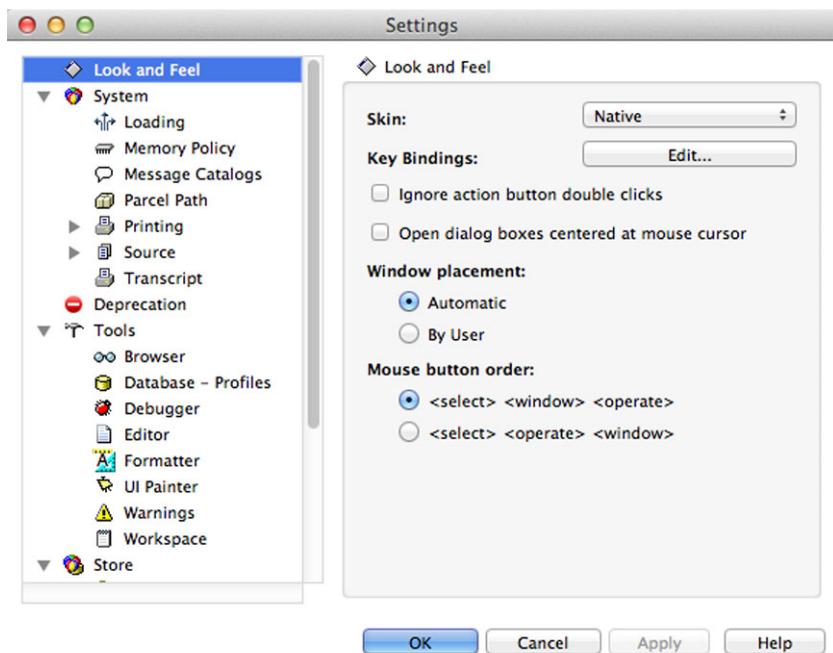


Fig. 5.4 VisualWorks configuration screen

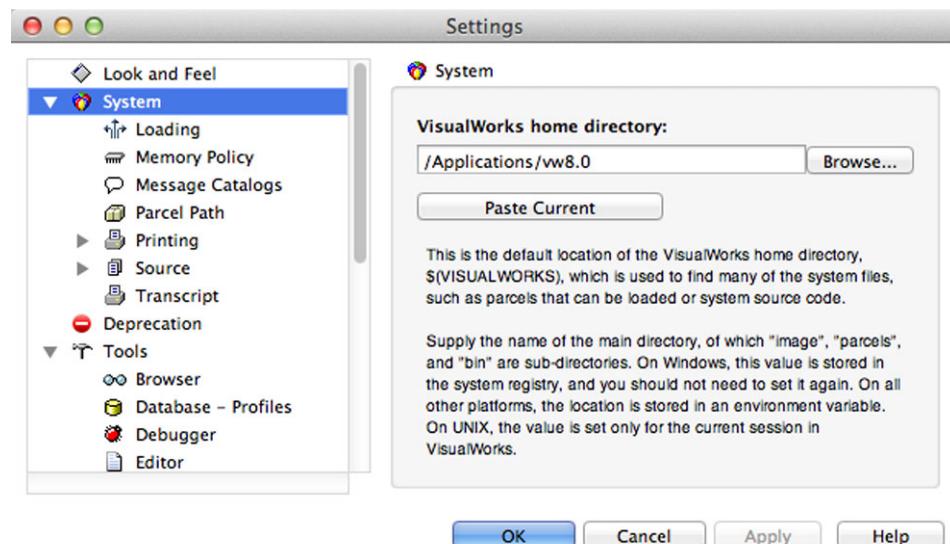


Fig. 5.5 The VisualWorks home directory

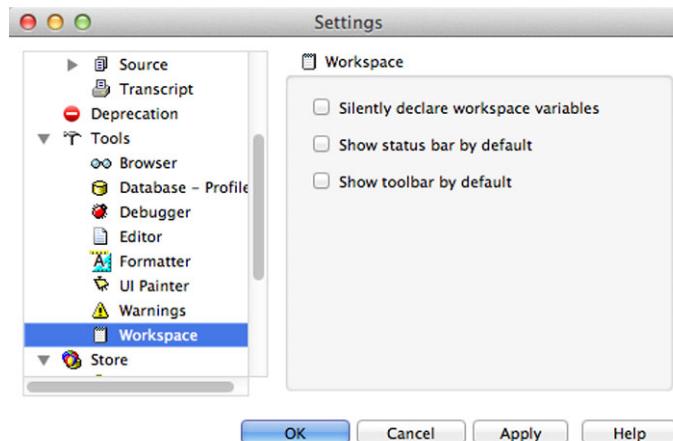


Fig. 5.6 Workspace settings

attempted by beginners. There's an important exception, though, as far as the behaviour of the workspace tool goes.

In order to work on these settings, select the **Workspace** entry in the left panel of the **Settings** window, as shown in Fig. 5.6. Uncheck all three options. The effect will be a workspace that looks and behaves like the examples used in this book. Above all, the "Silently declare workspace variable" option, when activated, can easily lead to confusion. If a variable is used in a program in the workspace without having first been declared, it will be introduced without any action on the user's part as a so-called *workspace variable*. This is a peculiarity of VisualWorks that does not occur in other development environments. When this option is not checked, the user is always prompted to say what should happen to an undeclared variable. In other words, the user always sees what is happening.

5.3.3 Using Transcript

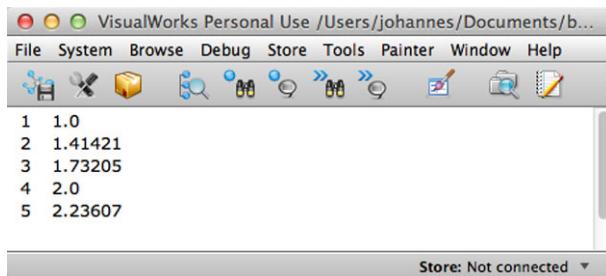
Transcript is a global variable that points to an instance of the class `TextCollector`. You can send a series of text-output messages to this object. For example, enter the following in the workspace:

```
Transcript show: 'Smalltalk is wonderful!'
```

When you use the **Do it** command to execute the line, the text sent as an argument to the `show:` message appears in the Transcript pane.

Output always appears sequentially in Transcript, that is, each new output is written directly after the previous one. Here is a quick overview of the most important messages

Fig. 5.7 Square roots of the numbers 1 to 5



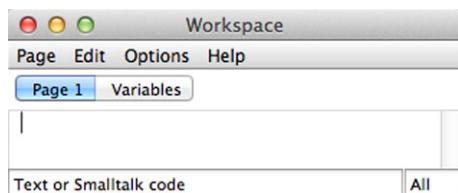
that Transcript understands:

show: aString—The character string aString is written to Transcript. For longer texts, an automatic line break occurs.
 tab—The output jumps to the next tab stop.
 space—A space character is output.
 cr—A carriage return occurs.
 clear—The contents of the Transcript pane are deleted.
 print: anObject—The textual representation of the object anObject is output to the Transcript pane. This message has the same effect as show: anObject printString. In that case, the printString message creates the string representation of the object anObject, because the show: message always requires an instance of the class String as an argument.
 nextPut: aCharacter—This writes the character aCharacter in the Transcript pane.
 nextPutAll: aString—This has the same effect as show: aString.

The following Smalltalk program calculates the square roots of the numbers 1 to 5 and creates an output in the form of a table (see Fig. 5.7).

```
Transcript clear.
1 to: 5 do: [:i |
    Transcript
        print: i; tab;
        print: i sqrt; cr].
```

Lines 4 and 5 make use of one of Smalltalk's syntactic capabilities, one that we have so far not used—the so-called *cascade of messages*. If you want to send several messages sequentially to the same receiver, it is only necessary to write the receiver once, while separating the messages to be sent by semicolons. That means that the expression

Fig. 5.8 A simple workspace

```
Transcript
    print: i; tab;
    print: i sqrt; cr
```

has the same effect as the following sequence of expressions:

```
Transcript print: i.
Transcript tab.
Transcript print: i sqrt.
Transcript cr
```

5.4 Workspace

From the very beginning, we've gotten to know the workspace as a tool that we can use to enter and test Smalltalk programs. At this point, we'll add a few comments on specific characteristics of the VisualWorks workspace.

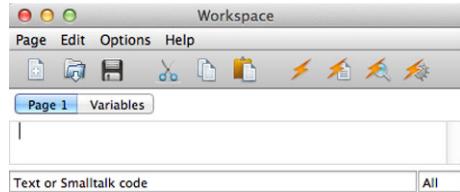
You can create a new workspace window by clicking **Tools → Workspace** on the **Launcher** menu. Several workspace windows can be open simultaneously. If you have configured the workspace settings as recommended in Sect. 5.3.2, a new empty workspace window appears as shown in Fig. 5.8. You will find similar workspaces in other development environments. In those cases, though, some of the capabilities of the VisualWorks workspace are not activated or are not as easily accessible. Since we make no use of them, we'll make just a few short comments about them.

When you have activated all of the workspace options, as shown in Fig. 5.6, a workspace like the one in Fig. 5.9 appears. Much like in the launcher window, a row of buttons appears beneath the menu bar that let you perform the most commonly used menu functions.

You can also see that a workspace can contain several different tabbed windows. One of them contains the workspace variables mentioned above if any have been declared.

Finally, underneath the text-input field, you can see a status line that contains information about the type of workspace page that has been selected, along with the *namespaces* imported by the workspace. Section 7.1.1 will describe *namespaces* in greater detail.

Fig. 5.9 A “complete” VisualWorks workspace



If you have a two- or three-button mouse, you can almost entirely avoid using the menu bar. By right clicking in the workspace, you can display a context menu from which you can access all of the workspace’s important functions. There are two exceptions: The only way you can save the contents of a workspace page is by selecting **Page → Save** from the menu (or by using the corresponding button), and the same is true for the command **Page → Open** when you want to open a text file as a workspace.

Four options are available for evaluating a series of Smalltalk expressions in the workspace, that is, for having a VM run the program, all of which have been used in previous chapters.

Do it evaluates—more or less silently—the sequence of expression selected in the workspace. That is, there is no output of the results of the sequence unless the Smalltalk code calls for such an output, for example, as output in the Transcript or by opening a dialogue window as occurred several times in Sect. 2.2.2.

Print it also evaluates the selected sequence of expressions. In addition, it writes to the workspace the textual representation of the object that is the result of the last expression in the sequence.

Inspect it evaluates the selected sequence of expressions and starts an Inspector for the object that is the result of the last expression.

Debug it evaluates the sequence of expressions and starts the bugger when it sends the first message. We’ll discuss this more in Sect. 5.6.

Of course, all of these options are possible even with the kind of simple workspace shown in Fig. 5.8. Since we won’t be using the expanded capabilities, we’ll limit our discussion to using the simpler workspace.

5.5 Inspector

In Sect. 3.1.4, we already described the use of Inspectors for examining the internal life of an object. At this point, we’ll add a few supplementary points that can be useful for your work in practice.

First off, let’s look again at the Inspector window from Fig. 3.8. Among other things, it shows the two instance variables `origin` and `corner` for the `Rectangle` object. Both of them point to a `Point` object. In order to inspect the object that is bound to an instance variable, select the variable and then select the function **Dive → yourself** from

Fig. 5.10 Diving into the corner component of a Rectangle object

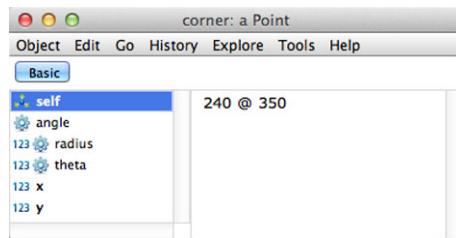


Fig. 5.11 Inspecting the y coordinate of the point in Fig. 5.10

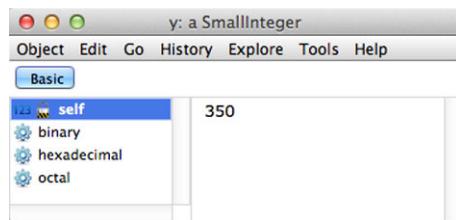
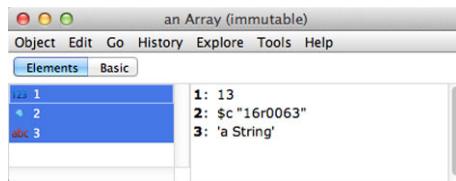


Fig. 5.12 Inspecting an array



the context menu.² The Inspector changes its appearance (see Fig. 5.10). The window title, “corner: a Point”, indicates that the corner component of the rectangle is currently being inspected, and that the component is a Point object. At this point, it’s possible to dive even deeper into the object structure, for example, by selecting the y coordinate and again selecting the menu item **Go → Dive**. The Inspector now shows the inner life of an object of the class SmallInteger (see Fig. 5.11).

By using the menu item **Go → Back**, you can rise up step by step from within the object structure. By clicking twice, you can return to the view from Fig. 3.8.

Inspecting Collection Classes

One frequently wants to use the Inspector to examine the components of a collection class, an array, for example. At this point, we’ll also learn another option for starting an Inspector. One can send any object an `inspect` message, which will start an Inspector for that object. If you use **Do it** to execute the following Smalltalk expression in the workspace, the window shown in Fig. 5.12 appears.

```
#(13 $c 'a String') inspect
```

²You can do the same thing by selecting **Go → Dive → yourself** from the menu bar.

You can see that the Inspector in the **Elements** tab presents a simplified view of the structure of a collection. In place of the instance variables visible in the left field, the indexes of the three components are shown, whose contents can by clicking then be viewed in the right field.

One can also use this view to rearrange the elements in the collection by dragging and dropping them with the mouse.

In general terms, an Inspector can also be used to manipulate the values of instance variables of an object. This is a technique that can occasionally be useful when debugging a program, but we will not describe it here in any further detail.

5.6 The Debugger

Section 4.1 provided an initial glimpse into the construction and use of the debugger. But it's only the practical exercise of searching for errors in complex applications that you can learn its true usefulness.

An important application of the debugger is in the step-by-step execution of methods. We'll pick up this technique in Chap. 9, because to understand this method of using the debugger, it helps to have knowledge of the structure of classes and their methods, which will not be treated until upcoming chapters.

5.7 System Browser

Among the most important tools of a Smalltalk development environment are various so-called *browsers*. Their purpose is to browse through the class library, which we could also call the Image. This is an important task, because a large part of the work of object-oriented programming involves reusing—to the greatest extent possible—existing partial solutions in the form of existing classes in the Image to solve a given problem. Such class libraries can be extensive, and so one needs effective tools to obtain an overview over what's already there. But just using the tools does not yield such an overview. One needs a lot of experience and practice to be able to home with some degree of accuracy on the classes and methods useful in solving the problem.

In VisualWorks, you start the browser from the launcher by clicking **Browse → System** from the menu or by clicking the corresponding button.



Figure 5.13 shows the System Browser with its start window in which, in the large lower pane (with the **Overview** tab), a short explanation of the function of the System Browser is displayed. This explanation uses important concepts like *package* and *namespace*, which will be described in detail in the following pages. Figure 5.14 shows the structure of the

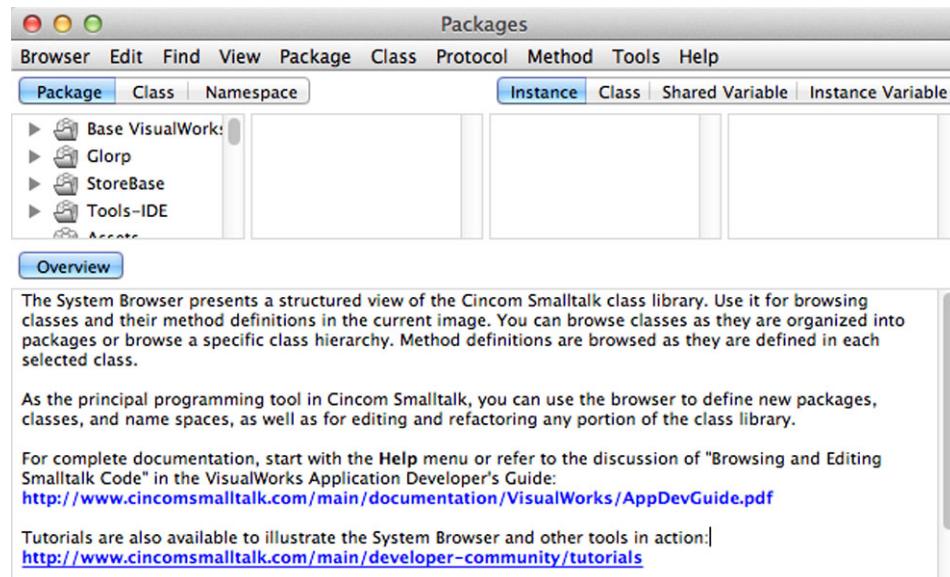


Fig. 5.13 The System Browser in VisualWorks

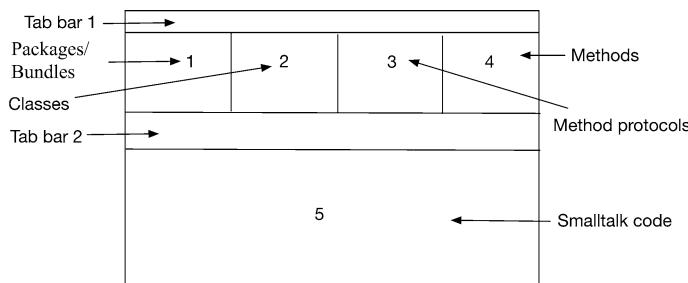


Fig. 5.14 Breakdown of the system-browser window

system-browser window.³ Field 1 shows the list of *packages* and/or *bundles* that exist in the Image.

Packages

The Smalltalk programming language itself has no additional structural level above classes. At bottom, a Smalltalk program consists of a collection of classes. But the VisualWorks class library alone contains hundreds of classes. And, of course, it's very hard to get any kind of an overview. To come to terms with this problem, the famous Blue Book

³For this and all other representations of the system browser, we are assuming that the **Toolbar** and **Status bar** options have been deactivated in the **Tools** menu.

(Goldberg and Robson 1989), which is still the most important language reference, described an ordering schema above the class level, which is used especially in the browsers. According to this schema, thematically related classes are combined into *class categories*. That means, for example, that geometry classes like `Point` and `Rectangle` are part of the category `Graphics Geometry`. One still finds this structure in some Smalltalk development environments such as Squeak. Be aware, though, that this is purely a grouping instrument for classes within the browser. The Smalltalk language does not know what a class category is. One can think of class categories as stickers that have been stuck to classes. One can use them to force the browser to display all classes bearing the same sticker. Beyond that, class categories are not connected to any other functionality.

Beginning with Release 7, the VisualWorks System Browser no longer uses a class-category oriented view of the classes found in the Image. VisualWorks still uses class categories, but they have become practically meaningless. Instead, packages are now the most important concept for grouping together thematically related classes. For that reason, as a rule, a VisualWorks Smalltalk application consists of one or more packages, which can be further grouped together into bundles. Packages do not, however, need to be part of a bundle. Besides structuring classes in the Image, packages and bundles are units that are managed by VisualWorks's own source-code-management system, *Store*. This especially supports the work of development teams.⁴

In Fig. 5.15, the package `Graphics-Geometry` has been selected in Field 1. The package is part of the `Graphics` bundle, which is in turn part of the `Base VisualWorks` bundle. When a package has been highlighted in Field 1, Field 2 displays a list of the classes that are part of the package and Field 5 displays a package comment that briefly describes the purpose and contents of the package.

When you select a class in Field 2 (`Rectangle`, for example, as shown in Fig. 5.16), Field 3 displays a list of the so-called *method protocols* of the selected class. Method protocols refer to categories used to organise by theme all of the methods of a class. And what's true for packages and class categories is also true for method protocols—none of them are part of the Smalltalk programming language. They're all merely part of the browser intended to provide a better overview.

► **Terminology note:** In Sect. 3.1, we used the phrase *method protocol* in a slightly different sense, as the set of all instance methods and class methods for a class. Sometimes the term is also used to mean the set of all messages understood by the objects of a class. In this discussion, we are using method protocol to mean a single method category. The terminology related to these matters is unfortunately not uniform in the specialist literature. From this point on, we will use the term in the last sense, that is, a single method category.

⁴Chapter 17 contains some information about *Store*.

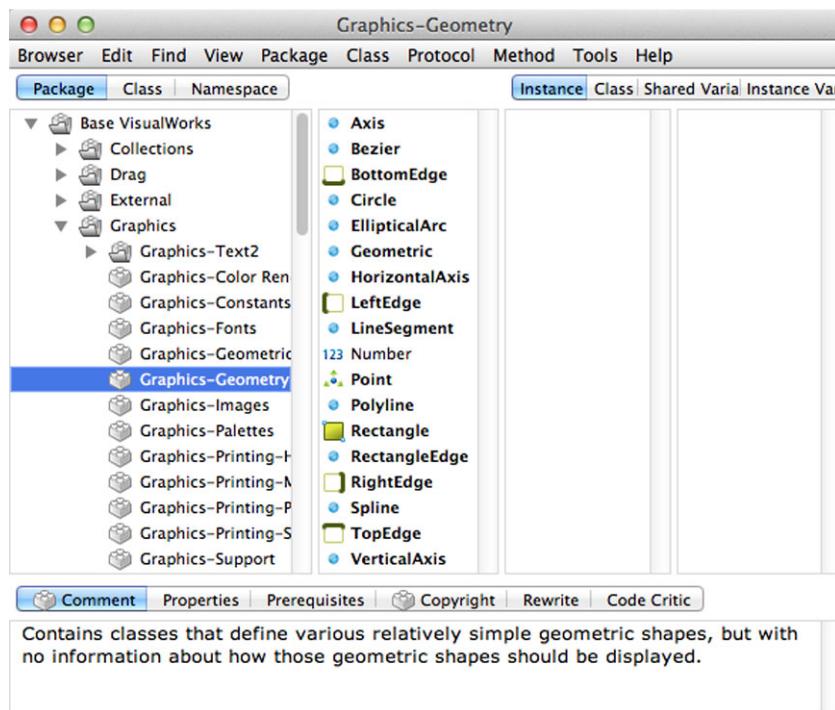


Fig. 5.15 Classes in the `Graphics Geometry` package

Until you select a method protocol in Field 3, Field 4 displays a list of all instance methods for all protocols (only partially visible in Fig. 5.16).

The **Comment** tab has been activated above Field 5. At the moment, Field 5 shows comments on the class, which is also only partly visible. It briefly describes the meaning and purpose of the class.

If one clicks the **Definition** tab, Field 5 displays the Smalltalk expression that resulted in the creation of the class `Rectangle` (see Fig. 5.17). In Smalltalk, classes are “normal” objects, which—like all objects—are created by sending messages to appropriate receivers. The receiver of the message is, along with `Smalltalk.Graphics`, a Namespace object. Chapter 7 describes namespaces and the details connected with creating classes.

At this point, we will just point to the fifth line

```
instanceVariableNames: 'origin corner '
```

which establishes that the instance variables `origin` and `corner` are available to the instances of the class, which was already dealt with in Sect. 3.1.4.

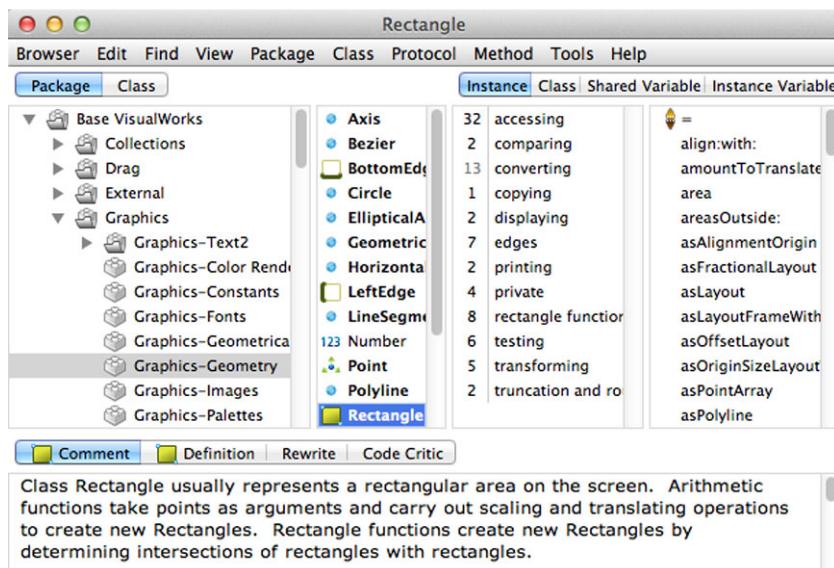


Fig. 5.16 Method protocol for the Rectangle class

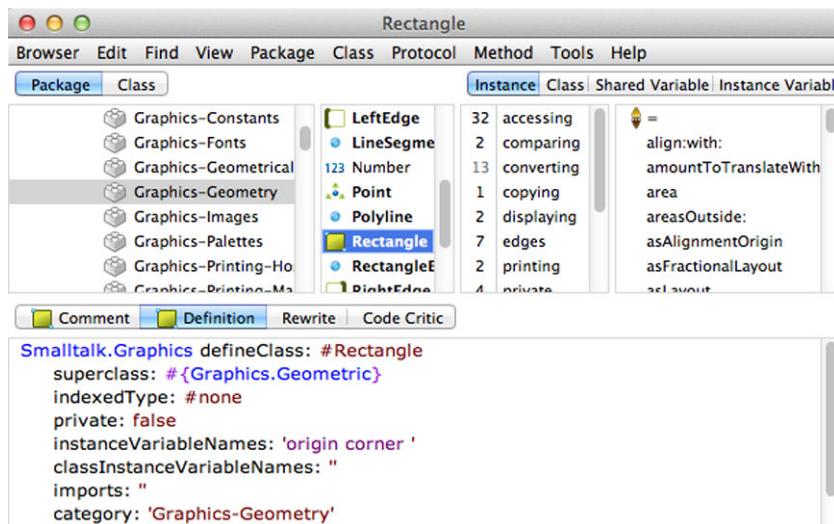


Fig. 5.17 Definition of the class Rectangle

If one selects one of the method protocols (for example, accessing), the browser displays Fig. 5.18. Field 4 now displays the names of the methods that belong to the selected protocol. At the same time, the content of Field 5 changes into a model that can

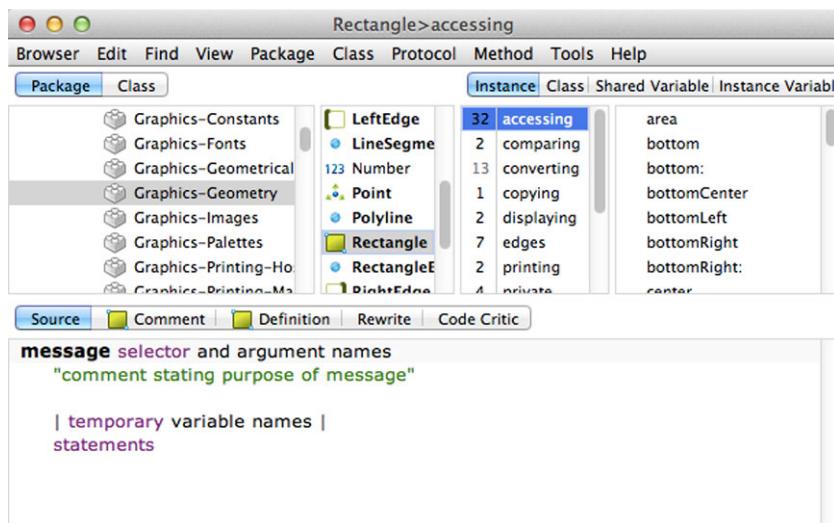


Fig. 5.18 Selecting a method protocol of the class Rectangle

be used as the starting point for the definition of a new method. We'll discuss that too in Chap. 7.

Finally, if one selects one of the methods in Field 4, Field 5 now displays the Smalltalk code that represents the implementation of that method, that is, the series of Smalltalk expressions that are executed when an instance of the class `Rectangle` receives a message with the same name. Figure 5.19 shows the implementation of the method `area`, which calculates the area of the rectangle by multiplying the height and width. For the moment, we'll postpone the question of how that actually occurs (see Chap. 6).

Above Fields 3 and 4 there are four tabs with the labels **Instance**, **Class**, **Shared Variable** and **Instance Variable** (see Fig. 5.13). Activating these tabs affects the contents of Fields 3 and 4. In previous examples and figures, it was always the **Instance** tab that was activated. In this state, the browser displays in Field 4 the *instance methods* of the class or of the protocol selected in Field 3. Instance methods are the implementations of the messages that are understood by the instances of a class. The message `area`, for example, is understood by an instance of the class `Rectangle` and responds with the measurement of the area. Moreover, we already got to know *class methods* in Sect. 3.1.4. These are messages that are understood by the class itself, and that are used primarily for creating objects. For example, in Sect. 3.1.4, we used the message `origin:corner:` to create a new instance of the class `Rectangle`.

When we activate the **Class** tab, the browser (see Fig. 5.20) displays in Field 3 a *class method protocol* whose name—`instance creation`—suggests that it gathers together methods used for instance creation. Field 4 of the method list shows the method `origin:corner:` selected, the implementation of which is shown in Field 5. Chapter 6 handles the details of these methods.

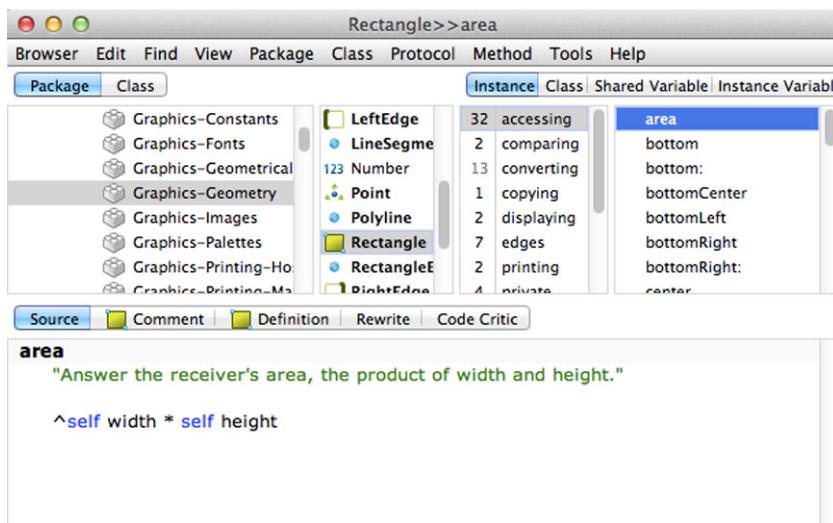


Fig. 5.19 The method area for the class Rectangle

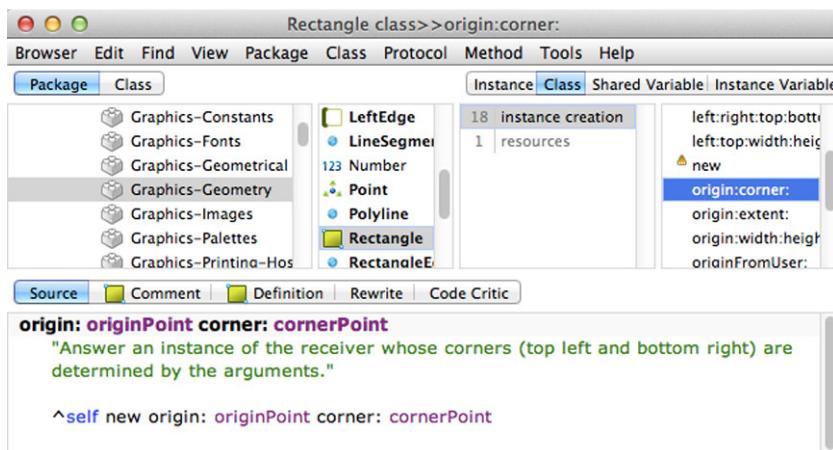


Fig. 5.20 Class methods of the class Rectangle

The third tab is used for inspecting or defining shared variables, for example, class variables (see Sect. 7.3). For now, we will not use shared variables, and so we'll spare any further discussion of them. If one activates the tab for the class Rectangle, Field 3 stays empty, that is, this class contains no class variables.

Activating the **Instance Variable** tab causes Field 3 to display the instance variables for the class selected in Field 2 (see Fig. 5.21). Selecting a variable limits the list of methods in Field 4 to those in which the variable is used.

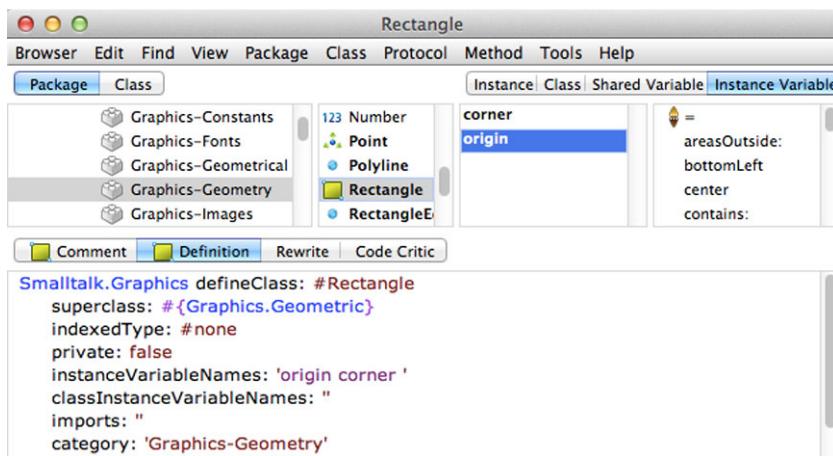


Fig. 5.21 Instance variables of the class Rectangle

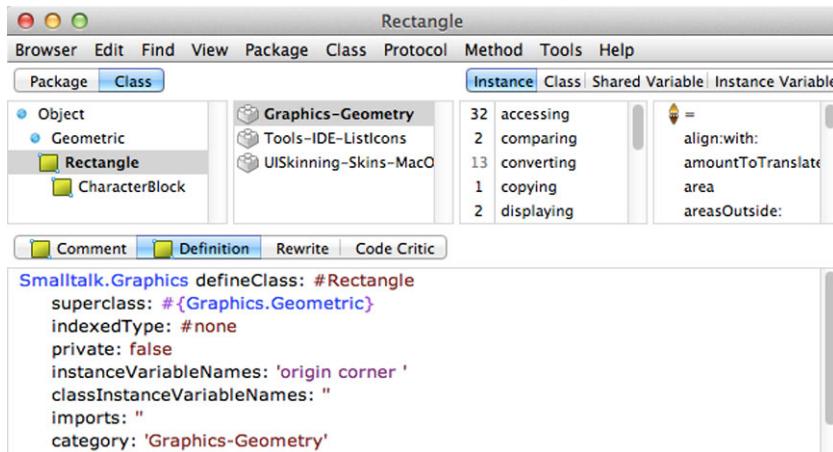


Fig. 5.22 The position of the class Rectangle within the class hierarchy

Additional System-Browser Functions

Like the other tools in the VisualWorks development environment, system-browser functions can be invoked either from menus in the browser window's menu bar or via context menus. Note, though, that each of the five fields has its own context menu. The context menu for Field 5 reflects the items in the **Edit** menu; those for Fields 1 to 4 reflect the **Package**, **Class**, **Protocol** and **Method** menus respectively. We will explain the individual functions when they are needed.

Activating the **Class** tab above Field 1 lets you display the position of the selected class within the class hierarchy. When you perform this selection, though, the contents of Field 5 remain unchanged (see Fig. 5.22).

At the end of this section, we remind you once more that you can use the System Browser to inspect and modify all classes in the Image and their methods. The exception to this rule are a few so-called primitive methods, which for reasons of efficiency are directly implemented in the VM. Note, though, that the development environment can become totally unusable if the programmer makes errors in modifying classes and methods that are important for the functioning of the development environment. For that reason, it is imperative that you always be able to restore your environment from the original Image.

In Chap. 3, in connection with the discussion of object creation, we said a little bit about the role of classes. With the assistance of the System Browser, Chap. 5 shed some light on the structure of a Smalltalk class and the way it's embedded in the class library. Now it's time to answer some of the open questions left from those chapters. For that purpose, we are going to use the class `Circle`. As we do so, however, new questions will arise, the answers to which will be reserved for the following chapters.

Examples of the unanswered questions include:

1. Section 5.7 mentioned that the way a class is embedded in the class hierarchy could be determined from the hierarchy view. But what is a class hierarchy?
2. In Chap. 2, we learned that the whole number 2 is an instance of the class `SmallInteger`, and that it understands the message `sqrt`. But why is there no method called `sqrt` in the method protocols of the class `SmallInteger`?

In the next section, we will answer these questions. After that, we will examine a method implementation somewhat more closely in order to provide a basis for supplying methods to some of our own new classes.

At the end of this chapter, we'll conduct a thought experiment that will emphasise once again the meaning and the advantages of information hiding in object-oriented programming. The implementation of the class `Circle`, which is available in the Smalltalk class library, will be modified in such a way that, when looked at from the outside, no change in behaviour will be visible.

6.1 Class Hierarchies and Inheritance

As was explained in Sect. 5.7, when we activate the **Class** tab in the System Browser, we see the hierarchical view of the class. Figure 6.1 shows the hierarchical view of the class `Circle`. The following is true in Smalltalk:

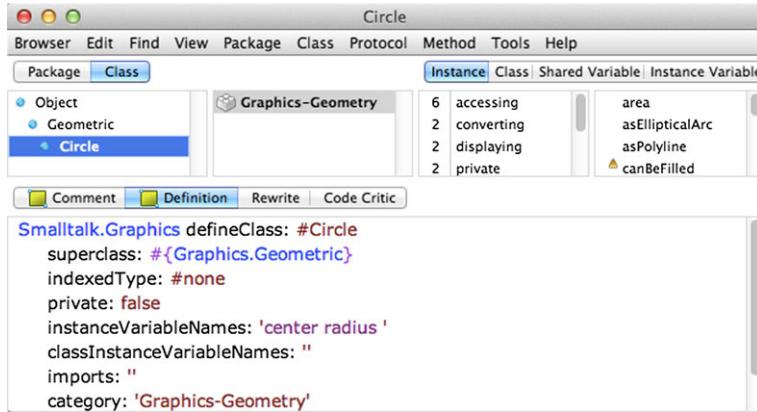


Fig. 6.1 The position of the class `Circle` in the class hierarchy

Each class has exactly one *superclass* from which it is derived. The only exception to this rule is the class `Object`, which has no superclass.

We can interpret Fig. 6.1 as follows: The class `Circle` has the superclass `Geometric`. Or conversely, the class `Circle` is a subclass of the class `Geometric`. And that class, in turn, is a subclass of the class `Object`. And so the class `Circle` may be said to be indirectly a subclass of `Object`. We say that a class is *derived* from its superclass. Notice too that the class `Circle` does not have a subclass. In the hierarchical view, subclasses are shown indented from their superclasses.

Another Smalltalk rule is:

Every class is directly or indirectly derived from the class `Object`. In other words, all classes are components of a common *class hierarchy*, which is presented as a tree.

In the System Browser, it's of course also possible to view the hierarchical view of the class `Object`. The class is located in the package *kernel objects*. The browser displays the entire class hierarchy of the current Image. Because of the large number of classes, this is a confusing and hardly useful representation. Instead, Fig. 6.2 shows a small section of the class hierarchy that shows the branching structure in a more useful way.

What does this hierarchical class structure mean? There are two answers to this question:

1. The class hierarchy represents a system of classification with the idea of ordering the classes into a graded structure that makes the system easier to comprehend.
2. One class *inherits* the structure (in the form of instance variables) and behaviour (in the form of methods) of its superclass. In that way, the structure and behaviour of a superclass can be *reused* in its subclass.

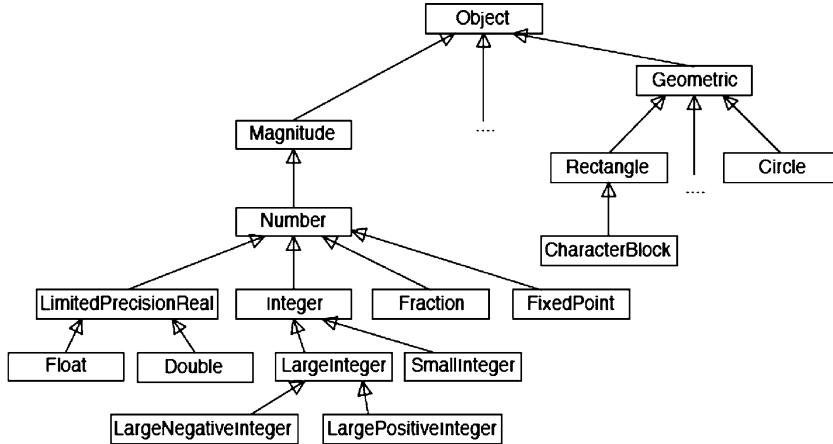


Fig. 6.2 A section of the Smalltalk class hierarchy

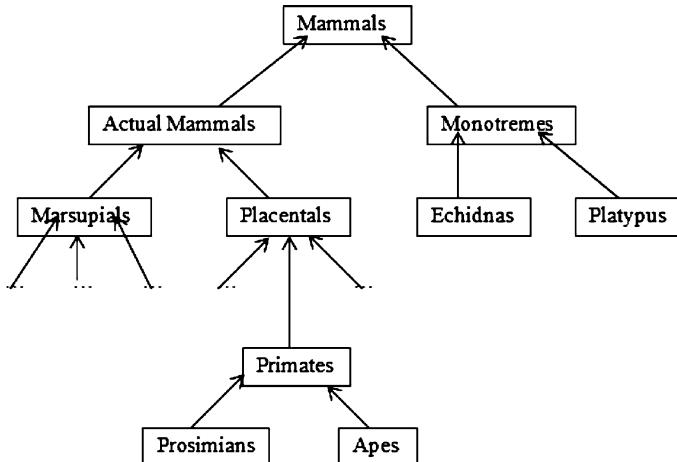


Fig. 6.3 A section of the taxonomic classification of mammals

Other areas of science also use classification schemas. It's always useful at this point to draw a parallel to the system of animal taxonomy that biology uses. For example, when classifying mammals, one could represent their class hierarchy in Fig. 6.3. The relationship between two classes represented by an arrow is in the form “is a.” An ape is a primate. A primate is a placental, etc. That means that an ape is a placental and, ultimately, a mammal.

Applying this to the Smalltalk class hierarchy in Fig. 6.2, this means that an instance of the class `SmallInteger` is an `Integer`, a `Number`, a `Magnitude` `Object` and finally an `Object` object.

Schemas of this kind, divided into class hierarchies, have at their root the application domain for which a program is to be developed. Ideally, such a structure exists in harmony with the concept of reuse, which was mentioned in the second answer to the question we posed earlier. To illustrate, let's look at the portion of the Smalltalk class hierarchy that contains various types of numbers. The class `Number` can be considered mathematically as the top classification for all types of numbers. Logically, then, members of the classes `Fraction` and `LimitedPrecisionReal`¹ (representing real numbers) are treated as special kinds of numbers and are placed in the class hierarchy as subclasses of `Number`. From a strictly mathematical viewpoint, however, this would lead to a different class hierarchy. In that case, the class `Integer` would be a subclass of `Fraction`. For technical reasons, the application-oriented viewpoint is not carried over into the isomorphic class hierarchy (see Chap. 8).

Within the class `Number`, methods are implemented that can be applied to all types of numbers represented by the subclasses of `Number`. Examples of these include a variety of mathematical functions including logarithms, square roots and trigonometric functions. These functions can also be applied to instances of the subclasses, because the behaviour of the superclass is inherited by its subclasses. For that reason, the following Smalltalk expressions:

```
4 sqrt
(12/3) sqrt
4.0 sqrt
```

all produce 2.0, which is the correct result. The message is sent in the first line to an instance of the class `SmallInteger`, in the second to an instance of the class `Fraction` and in the third to an instance of the class `Float`. None of the method protocols of these classes contain the method shown in the expressions. In this case, SmaViM searches upwards in the class hierarchy until it finds a class that contains the required method. In this case, that is the class `Number`. It is in this way that subclasses inherit the behaviour of their superclasses. This means that the program code for calculating a root can be created a single time in the class `Number` and by the process of inheritance be reused in `Number`'s subclasses.

This also answers the second question that we posed at the start of this chapter—how is it that objects of the class `SmallInteger` understand the message `sqrt` even though its method protocols do not contain a method with that name?

When SmaViM searches through the class hierarchy for a method suitable for a message and arrives at the class `Object` without having found the method, it aborts the execution of the program with an exception and displays the error message that we already saw in

¹In fact, instances of these classes represented only a very coarse approximation of real numbers.

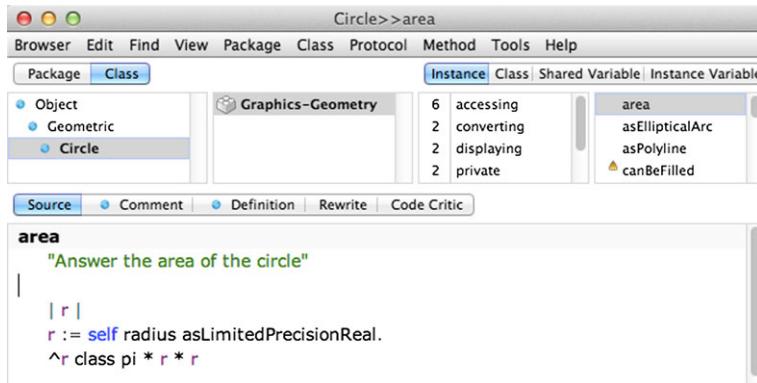


Fig. 6.4 The `area` method of the class `Circle`

Fig. 3.4. Be aware that SmaViM searches through the classes strictly in the order of the class hierarchy, that is, it looks from one class to its superclass, then to that class's superclass, until it finally reaches the class `Object`. Furthermore, the inheritance principle applies not just to the structure and behaviour of the instances of a class, but to the class itself. That means that a class also inherits its class methods from its superclass.

For reasons dealing with the way Smalltalk is implemented (which we won't go into here), this is not actually true. However, for the sake of simplicity, imagine that the class `Object` contains a class method `new`, within which the knowledge has been programmed of how to create a new object in the memory of a VM. That means in particular that every new class understands the message `new`, since each class is derived, directly or indirectly, from `Object`.

Finally, it needs to be pointed out that inheritance is one of the most important constituent principles of object-oriented programming.

6.2 Implementing Methods

Now we are going to examine the design of a method definition such as one might find in the System Browser or undertake oneself if one wanted to add new classes and methods to the Image. We've already seen an example of this in Sect. 5.7, when we looked at the method `area` for the class `Rectangle`, which appears in Fig. 5.19 on p. 94.

The class `Circle` also contains a method `area`, which is, of course, implemented differently from that for `Rectangle`. But before we examine this method in greater detail, let's first take another look at the definition of the class `Circle` shown in Fig. 6.1. We can tell from the instance variables that a circle is defined by its centre and its radius.

Figure 6.4 shows the implementation of the method `area`. The first line contains the so-called *message* or *calling pattern*, which shows how the message corresponding to the method is to be written when it is to be sent to the object. In this case, it's a unary message,

and so the message pattern consists of only the *message selector*. The next lines usually contain comments that describe the task and the result of the method.

Following the comments comes the declaration of temporary (local) variables should they be required. As we've already seen in the workspace, temporary variables are declared between two pipes (!).

Last come the Smalltalk expressions that are required for the implementation of the task, that is, for its algorithm. This part is also referred to as the *method body*.

The Meaning of the Pseudo-variable `self`

The first line of the method body

```
r := self radius asLimitedPrecisionReal.  
^r class pi * r * r
```

assigns the radius of the circle to the local variable `r`. In the second line, the radius is squared and then multiplied by π .

To understand exactly what is happening, let's look for a moment at the way the method `area` might be called in the workspace:

```
| circle area |  
circle := Circle center: 100@100 radius: 50.  
area := circle area
```

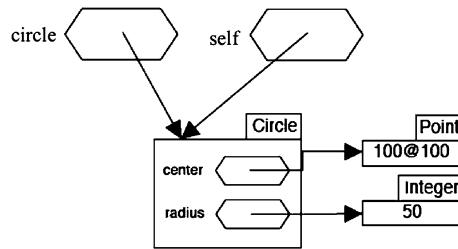
First, a circle is created by specifying its centre and radius, and is then assigned to the variable `circle`. Next, the message `area` is sent to this object, and the result is assigned to the variable `area`. Sending the message `area` leads to the execution of the method with the same name of the class `Circle`.

Since the radius of the circle is stored in the instance variable `radius`, it is possible to access it directly from within the method `area`. Instead, in the first line, on the right side of the assignment, we see the expression

```
self radius
```

This sends the message `radius` to the same object that receives the message `area`. During the execution of a method for an object, the pseudo-variable `self` serves the purpose of sending a message within the method to precisely this object—in other words, to itself. For the duration of the execution of a method, the VM automatically binds the variable to the object that is the receiver of this method. In our example, this means that at the

Fig. 6.5 Memory status while executing the method area



moment that the execution of `area` begins, `self` is bound to the same object as the variable `circle`. Figure 6.5 illustrates this state of affairs. `self` is called a *pseudo-variable* because, in contrast to “normal” variables, you cannot use an assignment to change the reference within `self`.

The expression `self radius` sends the message `radius` to the `Circle` object; the message itself does nothing more than supply the value of the instance variable of the same name. One could have achieved the same effect by directly accessing the instance variable at this point in the method. In that case, the first line of the method body of `area` would look like this:

```
r := radius asLimitedPrecisionReal
```

In Sect. 6.3, we’ll explain why we use a method call here instead of the faster, direct accessing of the instance variable.

By the way, the message `asLimitedPrecisionReal` converts the `radius` object, which in our example is an `Integer` object, into a `Float` object so that it can subsequently (to put it as simply as possible) be more easily multiplied with the `Float` number π . Using the class method `pi`, the number π can be determined from the `Float` or `Double` classes. In the second line of the method `area`, the expression `r class` initially determines the class that the object bound to `r` belongs to (in this case, `Float`). The message `pi` is sent to this class, and the result is then multiplied by the `radius`.

Defining the Result Object of a Method Activation

The method `area` must now return this result, the content of the area of the circle, as the result of the message `circle area`. In Chap. 3, we learned that sending a message always results in the return of an object. And so one needs a way to be able to mark a result object within a method. A so-called *return instruction* performs the task. A return instruction is a Smalltalk expression that starts with the `^` character, which is called a *return operator* (see Fig. 6.4). When a return instruction is executed, it interrupts the execution of the method and sends back, as the result of the method activation, the object that results from the evaluation of the expression that follows the return operator. In our example, this is the result of the calculation $\pi * r * r$.

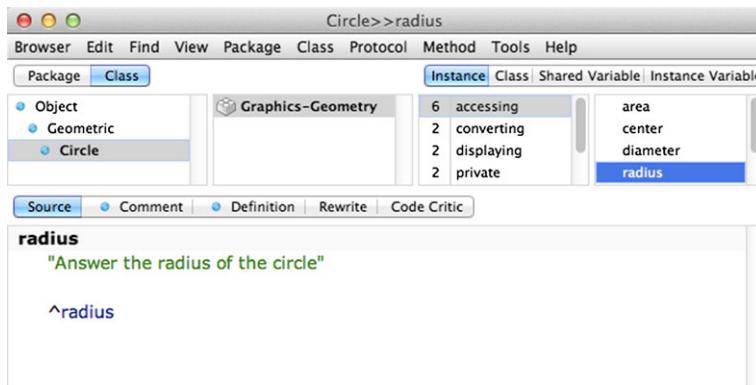


Fig. 6.6 Implementation of the method `radius`

Accessing Instance Variables

For the second example of a method implementation, let's look at the method `radius`, which is used in `area` to determine the radius of the circle. Figure 6.6 shows the method `radius` in the System Browser. That is, it produces the result of the evaluation of the expression

```
radius
```

as a result object. The expression uses the instance variable `radius`. This is possible—as well as necessary—here because instance variables are usually visible in all instance methods of the class that they belong to.

Since once again the return operator precedes the expression, the radius of the circle is once again the result of the activation of the method.

There are two schools of thought with regard to accessing instance variables of a class:

1. One should directly access instance variables whenever possible, that is, in the instance methods of the class.
2. Direct access to an instance variable is limited to a method with the same name as the variable (for example, `radius`)—a so-called *get method*—and to a so-called *set method* (for example, `radius:`, see below), where the value of the instance variable can be modified.

There's debate about which of the two variations is preferable. The first variation avoids a second method activation and is thus somewhat faster. The advantage of the second variation is that future changes in the structure of a class (changing the instance variables) can be carried out with little effort. If one follows this strategy consistently, then as a rule, it is necessary to define get and set methods for all instance variables. This pokes holes,

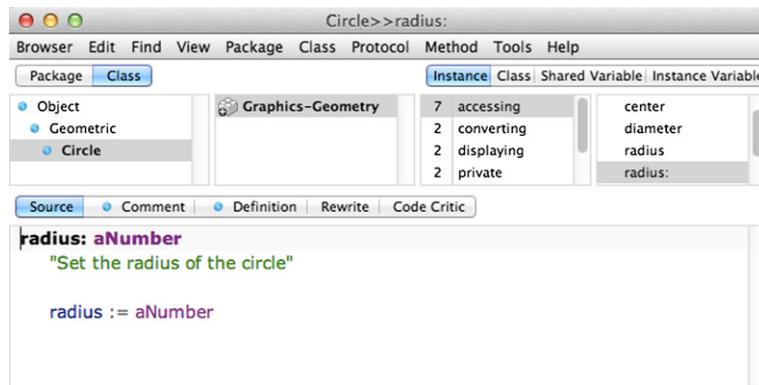


Fig. 6.7 Implementation of the method `radius:`

though, in the information-hiding principle of object-oriented programming, because—in Smalltalk at least—the use of methods cannot be limited by other objects. We will return to this topic in Sect. 6.3.

Defining a Keyword Method

Up until now, we've looked exclusively at unary methods. An example of a keyword method for `Circle` is `radius:.`. This makes it possible to define the radius of a circle with a new value that is sent as an argument along with the message. Figure 6.7 shows an implementation of this method. (Note: The original Image in VisualWorks does not include this method; it was added by the author.) The method might be used in the following manner:

```
| circle |
circle := Circle center: 120@200 radius: 35.
Transcript cr; show: circle radius printString.
circle := circle radius: 75.
Transcript cr; show: circle radius printString
```

In the fourth line, the radius of the circle is assigned a new value. Immediately before and after this line the current value is written into the Transcript. The values 35 and 75 appear sequentially.

This means that the `radius:.` message is sent along with the argument 75 to the object bound to `circle`. Within the implementation of the method one now needs a way to access the value of this argument. For this purpose, the first line of the method text, the message pattern, contains a placeholder—in this case, the name `aNumber`. Placeholders of this kind for arguments to be transmitted when the message is sent are also called *formal parameters*. The programmer can arbitrarily select the names for the placeholders.

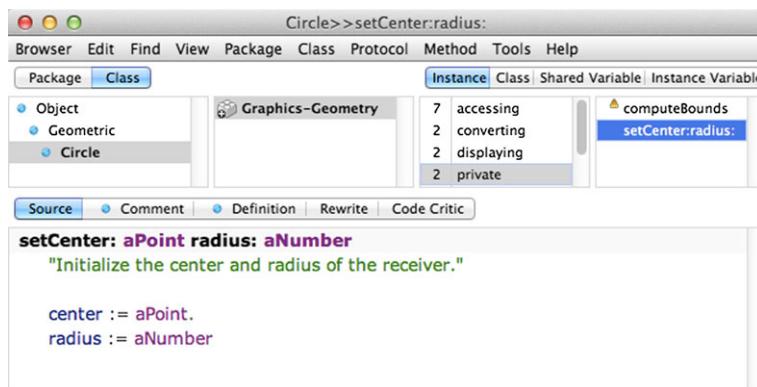


Fig. 6.8 Implementation of the instance method `setCenter:radius:`

In Smalltalk programming, though, it's customary to use names that suggest what kind (what class) the value of the argument should be.

During the running of the method, you can imagine the transmitted argument replacing the placeholder. In our example, every time the identifier `aNumber` occurs you can imagine the number 75 in its place.

You might notice that the method `radius:` does not contain a return operator. In such a case, the method automatically returns the receiving object as the result object as well. The purpose of the method here is not to determine a new result object, but rather for the receiving object to change its status, that is, the value of an instance variable. The `radius:` method causes the instance variable `radius` of the receiving object to receive a new value. A method without a return operator operates as though the last line contained the expression

```
^self
```

For keyword methods whose selector consists of more than one keyword, the calling pattern must make room for a placeholder after every keyword; the keyword stands for the argument to be transmitted at that point in the method. Figure 6.8 shows as an example the method `setCenter:radius:` of the class `Circle`.

The scope of validity of formal parameters extends over the text of the method and is thus equivalent to that of local variables. The names of the formal parameters must of course differ from one another, and must also be different from those of any local variables that are defined within the method. Furthermore, the names of both local variables and formal parameters may not conflict with names of variables used in the area of validity surrounding the method text. This would include, for example, the instance variables whose scope includes all instance variables of the class. That means that the name of a placeholder or a temporary variable may not be the same as that of an instance variable.

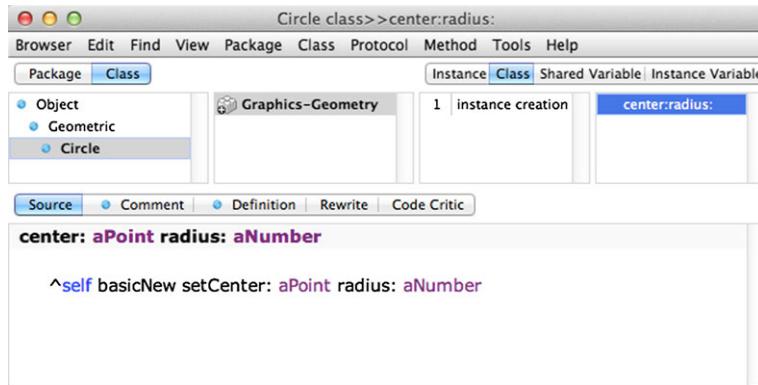


Fig. 6.9 Implementation of the class method `center:radius:`

The method shown in Fig. 6.8 permits you to redefine the centre and radius of an existing `Circle` object. This is an instance method, which should not be confused with the class method `center:radius:` used above, which served to create a new instance of the class `Circle`. In fact, one could have given the instance method `setCenter:radius:` the same name as the class method. SmaViM is able to differentiate between the various methods, because a class method always has a class as a receiver, while an instance method, in contrast, has an instance of the class as a receiver. Figure 6.9 shows the class method `center:radius:.`. This method first creates a new example of the class `Circle` using the partial expression

```
self basicNew
```

Thereafter, the two instance variables have the initial value `nil`, which stand for the undefined object. Note that in this method, the pseudo-variable `self` stands for the class `Circle`, which is, after all, the receiver of a class method. Once the instance has been created, the message

```
setCenter: aPoint radius: aNumber
```

is sent to it, which initialises the instance variables `center` and `radius`.

Defining a Binary Method

Finally, let's look at the definition of a binary method. As has already been explained in Sect. 3.1.1, binary methods are represented by operational symbols, which are represented by special characters. A method called `=`, which allows two circles to be tested for equality, might look like this:

```
= aCircle
    "Answer whether the receiver's species, center and
     radius match those of the argument, aCircle."
self species = aCircle species
ifTrue: [^self center = aCircle center
         and: [self radius = aCircle radius]]
ifFalse: [^false]
```

(Note: The original Image in VisualWorks does not contain a `=` method for the class `Circle`.) The calling pattern corresponds to a great extent with a keyword method with a single keyword, since a binary method also expects exactly one argument, for which a placeholder is provided in the calling pattern.

The method operates by checking whether the two circles (the receiver of the message and the one bound to the placeholder `aCircle`) are the same, that is, whether their centres and radii agree. The method can be logically applied, however, only when the object sent as the message's argument is an instance of the class `Circle`. The expression

```
self species = aCircle species
```

checks (simplifying somewhat) whether the two objects to be compared are both of the same type, that is, in this case, instances of the class `Circle`. If that is not the case, the result of the method activation is `false`. Otherwise the method checks whether the `center` and `radius` components of the two circles are equal.

6.3 Alternative Implementation of the Class Circle

Section 3.1 described information hiding—often called the encapsulation principle—as an inherent characteristic of object-oriented programming. The gist of this concept is that an object displays externally only a single behaviour, which is defined by the messages that it understands or the methods that underlie them. In contrast, the inner structure of the objects remains hidden to the outside observer. This makes it possible to change the inner structure without changing the externally visible behaviour of the objects. This ability creates an important prerequisite for the ability to revise implementation decisions after they have been made, without having to perform extensive adjustments in a complex software system consisting of many classes. It might be necessary, for example, to make such changes if the initial implementation concept should prove to be inefficient.

As long as the change that needs to be made affects only the structure of the objects of a class but does not change their behaviour, the modified class can replace the initial one without any problems. The next section illustrates the basics of this technique in the form

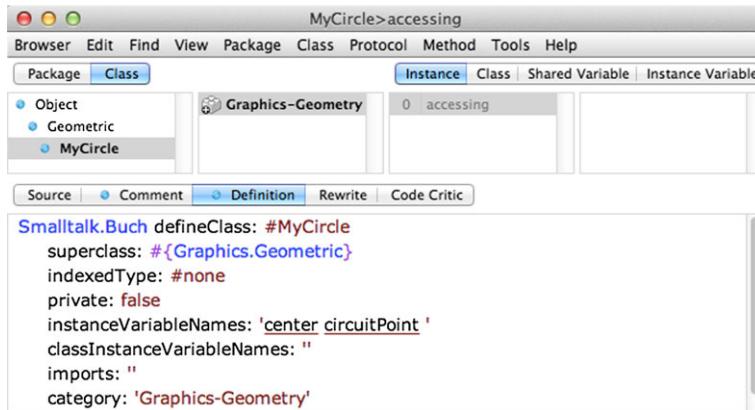


Fig. 6.10 The class MyCircle

of a thought experiment. We will, in fact, examine an alternative implementation of the class `Circle`.

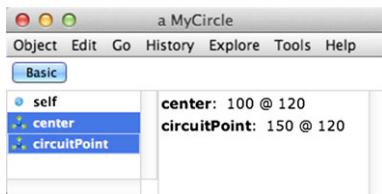
The Class `MyCircle`

The structure of the class `Circle` consists of the two instance variables `center` and `radius`. But it would also be possible to define a circle by its centre and a second point located on its circumference. In other words, we would keep the instance variable `center`, but replace the instance variable `radius` with a new variable that we might call `circuitPoint`. We're not interested in why someone would want to make such a change. There's certainly hardly any discernible advantage compared to the structure of the original class. But at this point, we're just trying to show that, in principle, it's possible to make such a change. Figure 6.10 shows the definition of the class `MyCircle` in the System Browser.

If you want the new class to actually replace the original class, it must have the same name. It's not recommended, however, to make direct changes to the original class, because it's useful to perform tests on the original class to compare its behaviour to that of the new class while you're in fact working on the new class. For that reason, it's better to perform the development work in the new class, and then rename it at the end.

Instances of the class `MyCircle` should behave exactly like those of the class `Circle`. Among other behaviours, it should still be possible to send the message `radius` to an instance of the class `MyCircle` in order to determine the length of the radius of the circle. The same method in the original class simply makes use of the relevant instance variable. That variable, though, is no longer available to the objects in `MyCircle`. The length of the radius must therefore be calculated as the distance between the points at the centre and on the circumference. To do that, we can change the programming of the method `radius` in the following way:

Fig. 6.11 An instance of the class MyCircle



```
radius
"Answer the radius of the circle"
^center dist: circuitPoint
```

The instance method `dist:` of the class `Point` calculates the distance between two points.

It must, of course, still be possible to create a new circle using the class method `center:radius::`. As we saw in the last section, the class method (see Fig. 6.9) uses the instance method `setCenter:radius::` (see Fig. 6.8), which must now be modified to accord with the new class structure:

```
setCenter: aPoint radius: aNumber
"Initialize the center and circuitPoint of the
receiver."
center := aPoint.
circuitPoint := aPoint + (aNumber@0)
```

In this case, the point on the circumference is calculated by adding to the centre point a point (`aNumber@0`) with the radius as its x coordinate and with a y coordinate of 0.

Now we can create instances of the class `MyCircle` in the same way that we created instances of `circle`. If we use **Do it** to execute the following expression in the workspace, the Inspector shown in Fig. 6.11 opens:

```
(MyCircle) center: 100@120 radius:50) inspect
```

And now we can also send the message `radius` to an object in `MyCircle`. Using **Print it** to execute the expression

```
(MyCircle center: 100@120 radius:50) radius
```

produces the correct output of `50.0`.

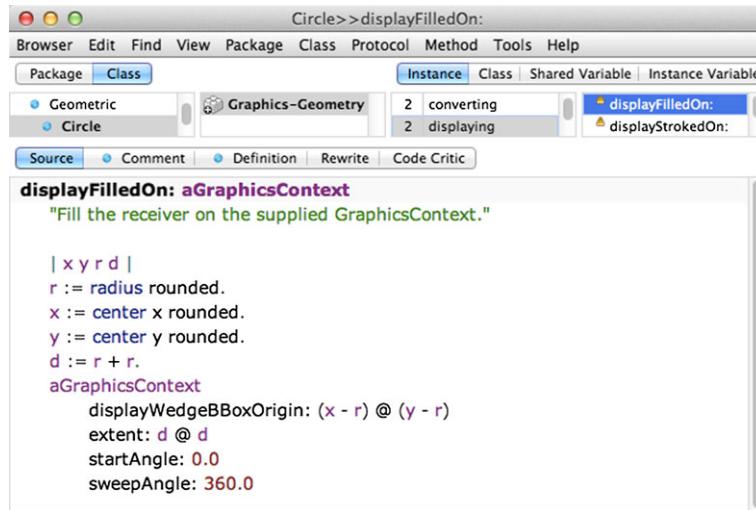


Fig. 6.12 Implementation of the method `displayFilledOn:` in the original class `Circle`

The behaviour with regard to creating new instances and to the reaction to the message `radius` is thus the same for both classes. It's now only necessary to examine the other methods of the class `MyCircle` to see if they make use of the omitted instance variable `radius`. By the way, this would not even be necessary if it were the general practice (except in the case of get and set methods) to avoid using instance variables when implementing methods, that is, for example, to use the message `radius` to determine the length of the radius of the circle. Once we've redefined the method `radius`, we would almost be done if all methods of `Circle` contained the expression `self radius` instead of `radius`. On this matter, though, the implementation of the class `Circle` in the original Image was not entirely consistent. Although direct access to the instance variables was avoided in most of the methods, there were two exceptions: the methods `displayFilledOn:` and `displayStrokedOn:` in the method protocol `displaying`. Figure 6.12 shows the method `displayFilledOn:.` In that case, the instance variables `radius` and `center` were used in the first lines. The author is not aware of a valid reason why the rule to use the relevant get methods was not adhered to.

The methods were modified to adhere to the general rule in the class `MyCircle`. (See Fig. 6.13, in which the relevant lines have been highlighted.)

No additional changes to the methods are required. Without bothering to prove the matter in detail, we can maintain that, after making the necessary changes to the class `MyCircle`, class's instances behave in exactly the same way as those of the original class `Circle`. We could eliminate that class and replace it with `MyCircle`.

In summary, we see that the strict encapsulation of the object structure makes it possible to replace one class with another. This has no effect on the other classes of an application, as long as the behaviour of the objects of the new class does not differ from that of the

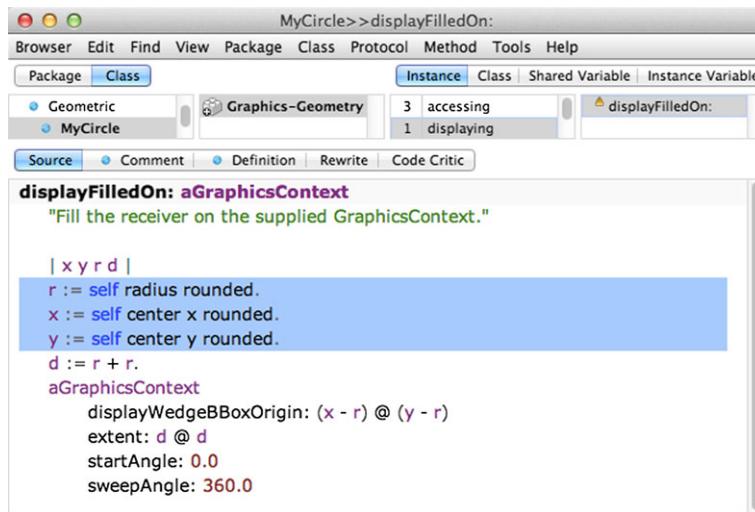


Fig. 6.13 Implementation of the method `displayFilledOn:` in the class `MyCircle`

objects of the old class, that is, as long as the new objects display the same reaction to the same messages. If, as described above, all methods of the class `MyCircle` are consistently matched to the structural changes necessitated by the instance variables, then this class can seamlessly replace the original class `Circle`. This is especially able to occur with little effort when the direct use of instance variables is limited entirely to get and set methods, whose only task, after all, is to access instance variables. We want to point out again the significance of the basic principle of object-oriented programming—that objects communicate with one another exclusively through sending one another messages, with no underlying assumptions about the internal structure of the various communication partners.

One of the central tasks in the development of an object-oriented application program lies in defining appropriate classes and their methods, the objects of which in their behaviour imitate real-world objects in an appropriate manner. In a commercial application, these objects might be products, contracts, bids and customers. In a geometric application, they could be lines, circles and rectangles. In a CAD¹ system for building machines, they could be screws, nuts and cogwheels or even complex objects like gears and engines.

The task of using program-design artifacts like classes and methods to accurately replicate real-world objects is very complex and requires an intensive dialogue between program developers on the one hand and experts in the various areas of application on the other. It might well be true that, in the development of complex software systems, most errors occur in this transformation of knowledge of the application into a software design; the errors often result from misunderstandings between the participating groups.

These questions of system analysis and software design are part of *software engineering*—an area of study that we will just lightly touch on.

First of all, one needs to know what engineering expertise is necessary for the definition of new Smalltalk classes. At the same time, we can't avoid examining the particularities relating to this expertise that one encounters in the development environment that's being used.

We'll consider the following matters using two case studies:

1. In Sect. 7.1, we'll re-examine the problem of currency conversion that we first considered in Sects. 2.1 and 2.2.
2. In Sect. 7.2, a small problem situation will be analysed and an object-oriented solution will be developed.

¹CAD = Computer Aided Design.

In Chap. 8, after we've introduced the topics of *class hierarchies* and *inheritance*, we'll return to the example of solving a quadratic equation (see Sect. 2.3).

7.1 Case Study: Currency Conversion

Let's imagine for a moment that we've been assigned the development of a program for a Web shop that operates internationally. Within this application, one encounters the problem of needing to display prices in different currencies, and thus having to convert prices from one currency to another. Thus, one of the program requirements is the ability to work with various "currency converters". We might regard them as instances of a class that in each case knows its rate of exchange.

These thoughts might lead to the following test program in the workspace:

```
| euroToDollar dollars |
euroToDollar := Converter withExchangeRate: 1.55.
dollars := euroToDollar convert: 227.0
```

In the second line, the message `withExchangeRate: 1.55` is sent to the class `Converter`. That should allow the class to create an instance that uses the exchange rate 1.55 to convert these amounts. In the third line, the `Converter` instance is used to convert 227 euros into dollars. Once our currency converter is finished, the evaluation of the test program via **Print it** should yield the value 351.85.

► **Note:** For simplicity's sake, we always show currency amounts as floating-point numbers. From a program engineering viewpoint, this practically counts as malpractice, since floating-point numbers² are intended for engineering or scientific calculations. They offer constant relative accuracy. With currency, though, one needs constant absolute accuracy (to hundredths, for example). Rounding errors that occur when calculating with floating-point numbers make our currency converter unsuitable for a currency trader who transfers billions from one currency into another.

In Smalltalk,² one could also avoid rounding errors by working with fractions (instances of the class `Fraction`) rather than with floating-point numbers. In that case, one would have to specify the exchange rate as `(155/1000)` instead of 1.55. But in that case, the converted amounts would also appear as fractions, which would then need to be converted into decimals for better legibility.

For a real commercial application program, though, it would make sense to show currency amounts not simply as numbers but rather as instances of a currency class, with which one could calculate with commercial accuracy.

²See Sect. 8.1.2 on this matter.

In order to convert among other currencies, one can always create additional instances of the class `Converter`.

In the following sections, we will gradually introduce each step to allow the program described above to run.

7.1.1 Creating a New Class

The preferred way to create a new class is via the System Browser. With VisualWorks, the first thing we have to determine is which package the new class will be created in. As a rule, one would create one or more packages for new classes specific to an application. Only rarely would it make sense to expand packages in the original Image with classes of one's own creation. Selecting one's own package also makes it easier to find the pertinent classes at a later date.

Creating a New Package

For our small application, we will simply introduce a package that we'll call `Currencies`. Selecting **New Package ...** from the **Package** menu of the System Browser opens a dialogue window in which one enters the name of the new package. At that point, the System Browser displays the window shown in Fig. 7.1. Field 5³ reminds the programmer that so far no comments have been added for this package.

Creating a New Namespace

Section 5.7 already explained that the program for creating a new class in VisualWorks sends an appropriate message to a namespace.

Namespaces are a special feature of the more recent versions of VisualWorks. Until Version 3, VisualWorks followed the model of other Smalltalk dialects and had only a single namespace, named `Smalltalk`. That namespace still exists, but should be used only in exceptional cases to create one's own classes.

Namespaces represent locations for identifiers, such as classes and global variables. That means that, within a single namespace, an identifier may be defined only once. In other words, within a namespace, identifiers must be unambiguous—a class name may be assigned only once. When only a single namespace exists, software development involving large teams can lead to problems, because the teams must agree before they begin assigning names. When several namespaces are available, one could, for example, assign one to each team. Name assignments would then be a matter of agreement only among the members of the smaller team. But one could also assign individual developers their own individual namespaces and thus prevent any conflict. You can find detailed instructions on the meaning and use of namespaces in VisualWorks in Cincom Systems (2014b).

³See Fig. 5.14.

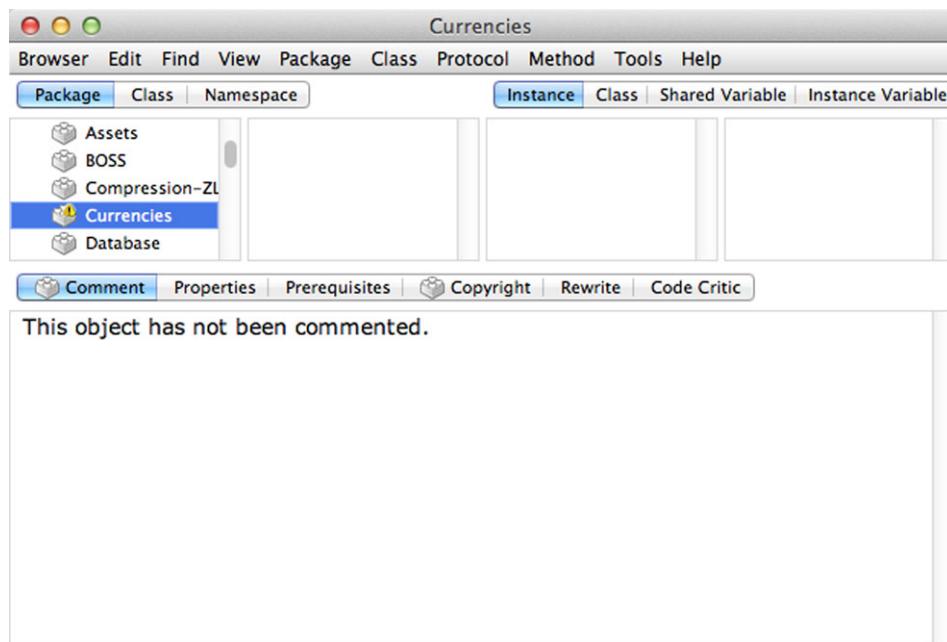


Fig. 7.1 After creating a new package

Before starting to develop an application, one should create at least one namespace. A new namespace is usually created as a sub-namespace for the namespace `Smalltalk`. For the small applications in this book, the definition of a single namespace is sufficient.

Before creating a namespace, you should be sure that you've highlighted the correct package in the System Browser. You can access the dialogue for creating a new namespace from the menu link `Class → New → Namespace`. In Fig. 7.2, we've already entered `CurrenciesNs` as the name of the namespace we're about to create. You don't need to make any other entries or changes in the dialogue box. After you click `OK`, the System Browser shows the definition of the new namespace, as shown in Fig. 7.3, where you can see that:

1. The namespace `CurrenciesNs` was created in the package `Currencies`; and
2. The namespace was defined by sending the message

```
defineNameSpace: #CurrenciesNs  
private: false imports: 'private Smalltalk.*' category: ''
```

to the namespace `Smalltalk`.

Creating the Class `Converter`

We have now performed all of the programming prerequisites necessary to create the class `Converter` in our package `Currencies` and in the namespace `CurrenciesNs`. Re-

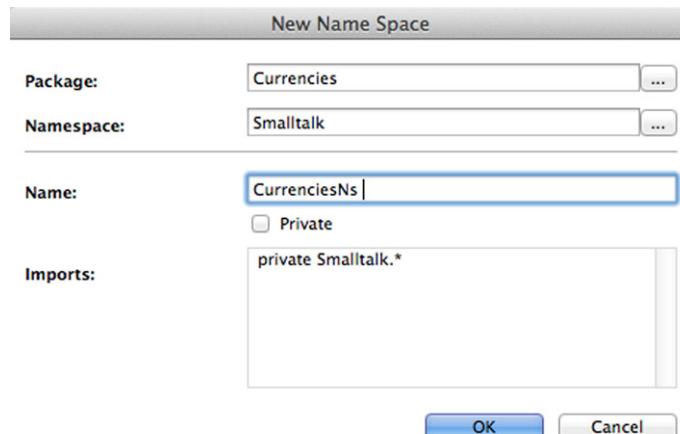


Fig. 7.2 Entering a namespace name

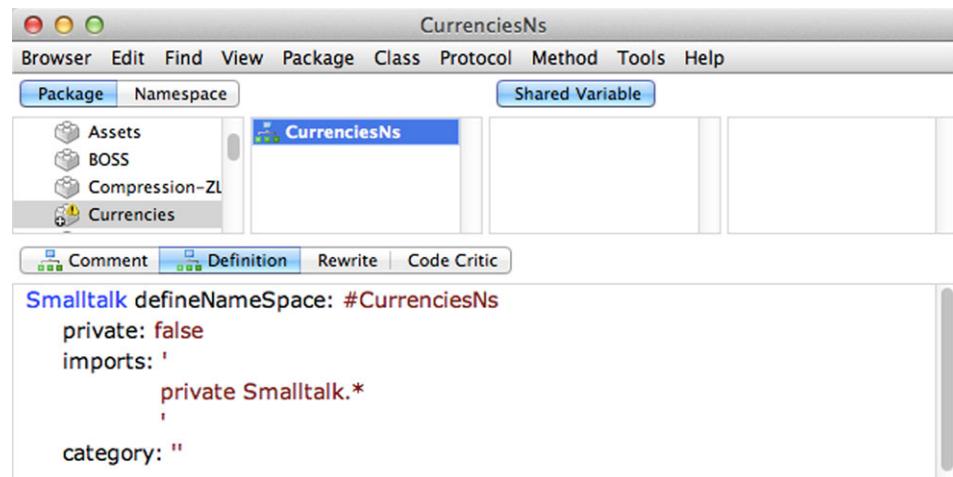


Fig. 7.3 Definition of the namespace CurrenciesNs

member that at the beginning of Sect. 7.1, we stated the requirement that the Converter instances should either know the various exchange rates or that they be provided for them as they are created. We can implement that requirement by specifying an instance variable `exchangeRate` when we define the class `Converter`. For now, instances of the class `Converter` have only a single property, the exchange rate.

Figure 7.4 shows the dialogue window for the creation of a new class. In order to display this window, highlight the desired package in the System Browser, then the desired namespace, then click on the menu item **Class → New Class...**

Review the following entries for our example:

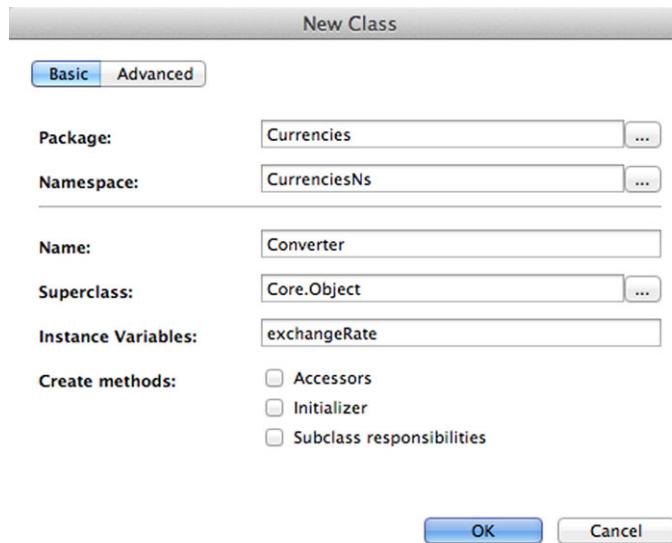


Fig. 7.4 Dialogue for creating a new class

1. Make sure that the entries for package and namespace are correct.
2. In the **Name:** field, enter the name of the class you are creating, in this case, `Converter`.
3. The **Superclass:** field already contains the class `Object` as the superclass,⁴ which remains unchanged.
4. In the **Instance Variables:** field, enter the names of any instance variables, in this case `exchangeRate`. If you enter more than one instance variable, separate them by spaces.
5. Finally, clear the three check boxes **Accessors**, **Initializer** and **Subclass responsibilities**. If you leave these boxes checked, several methods would automatically be created, which we'd rather not do at the moment.

After you click **OK**, the System Browser displays the definition of the class `Converter`, as shown in Fig. 7.5. Once again, Field 5 shows the Smalltalk message that affected the creation of the class.⁵ The receiver of the message is our namespace `Smalltalk`. `CurrenciesNs`. The message itself is a keyword message with the selector:

```
#defineclass:superclass:indexedType:private
  :instanceVariableNames:classInstanceVariableNames
  :imports:category:
```

⁴The entry `Core.Object` indicates that the class `Object` is defined in the namespace `Core`.

⁵There will be a warning that the variable `exchangeRate` is not referenced. It can be removed by hitting the Esc key.

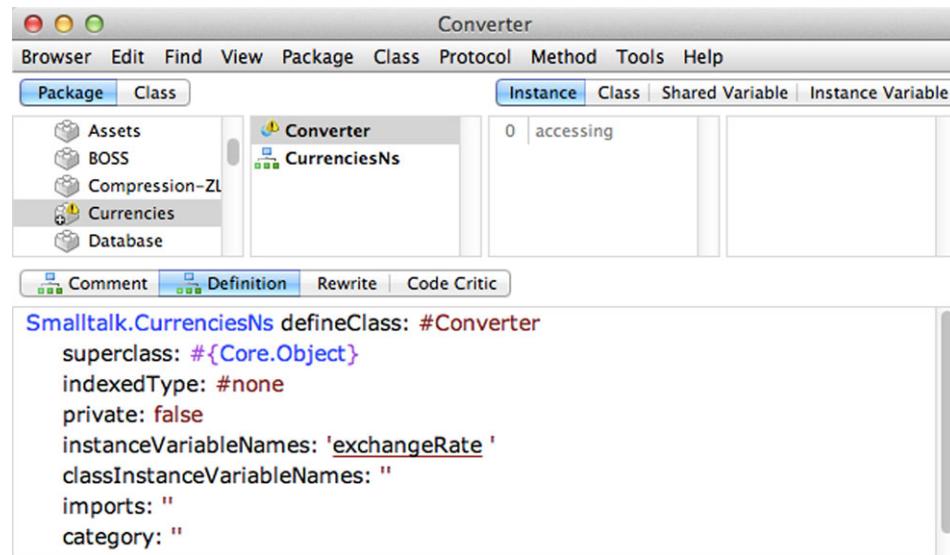


Fig. 7.5 Definition of the class Converter

This is a message with a total of eight keywords. We need to mention here that in Smalltalk—in contrast to other object-oriented programming languages—classes are not declared; rather, they are created as objects (classes are objects too!) when a program is executed. A message doesn't need to be sent from within the System Browser; it can actually occur anywhere that Smalltalk expressions are evaluated, for example, in the workspace or within a method. That means that it's possible to write Smalltalk programs that create new classes at runtime. The examples in this book, though, will not go so far as to show such an occurrence.

After the keywords, you must supply the following arguments in order to create a class:

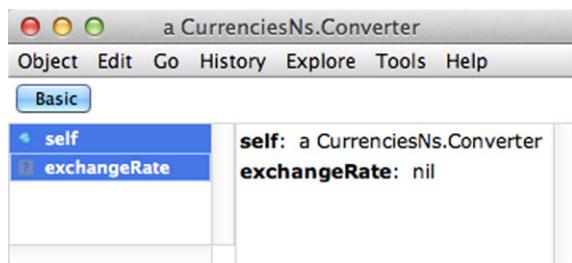
defineClass: The name of the new class as a symbol (see Sect. 3.2). In our example, that would be `#Converter`.

superclass: The name of the superclass of the class being created. The special syntax—with the name between curly braces—allows you to specify a namespace or a namespace path before the actual class name.

indexedType: The symbol `#none` as an argument indicates that our class has individually named instance variables. In the definition of the class `Array`, this space has the symbol `#objects`, indicating that instances of this class can vary in size (arrays can be created with any number of components) and that the components are so-called indexed instance variables that are addressed via a number (an index).

private: We leave this argument unchanged as well. An argument `private:true` prevents a different namespace from addressing this class.

Fig. 7.6 An instance of Converter



instanceVariableNames: The argument for this keyword consists of a character string containing the names of the instance variables, with the individual names separated from one another by a space. The order doesn't matter.

classInstanceVariableNames: Similarly, the next line could define so-called class instance variables. These are a class's private variables (not its instance variables). Only the class methods of a class can access them. We will not make use of them.

imports: This line can contain namespaces so that the identifiers defined in them can be used in the methods of the class `Converter` without specifying the namespace (that is, without specifying the namespace path).

category: This keyword is more or less a remnant from earlier versions of VisualWorks, when class categories were still an important organisational tool within the System Browser. They are now meaningless. (See Sect. 5.7.)

Once the new class has been created, we can immediately begin to create instances of the class `Converter` by sending the message `new`. Since the class method `new` was inherited from the class `Object`, this allows the expression

```
Converter new inspect
```

to be evaluated using **Do it**, which displays the Inspector window shown in Fig. 7.6. One can see there that the instance variable `exchangeRate` is bound to the undefined object `nil`.

By means of the class method `new`, the class method `Object` may be said to know how to create uninitialised instances of any class. This is a good example of the use of inheritance in the sense of the reuse of program code. What has once been implemented in the method `new` can now be used by all other classes. The encapsulation principle means that one doesn't need to worry about how to go about creating objects.

7.1.2 Individualised Class Methods for Creating Instances

From an application-oriented point of view, there should be no such things as instances of the class `Converter` created with `new`. That's because the instance variable `exchangeRate` has not been defined, and so an attempt to use such an instance for converting an amount of money must fail. It would probably be possible to send an appropriate message that would set the rate of exchange for a `converter` object after the fact, but it would certainly be preferable if the rate of exchange could be supplied at the same time the instance is created. And that's exactly what the expression `Converter withExchangeRate: 1.55` does in our little test program:

```
| euroToDollar dollars |
euroToDollar := Converter withExchangeRate: 1.55.
dollars := euroToDollar convert: 227.0
```

In order for this expression to be evaluated, we have to be sure that the message `withExchangeRate:` is understood by the receiver, that is, by the class `Converter`. In order for a class to understand a message, it must either itself have a method with the same name at its disposal, or else it must be able to access it because it has inherited the method. In this case, we have to define an appropriate class method in the class `Converter`.

The start of the method might look like this:

```
withExchangeRate: aNumber
    "creates an instance of the receiver with the
     rate of exchange aNumber"
```

The first line represents the calling pattern. In this example, the formal parameter that serves as a placeholder for the value transferred by the message (in the above example, the number 1.55) is called `aNumber`, which is supposed to indicate that a number is expected here as the argument. In general, though, the names of formal parameters can be arbitrarily selected.

The comment that begins in the second line briefly describes the purpose of the method. Note the way this is worded, "...an instance of the *receiver* ...". Since the message `withExchangeRate:` is sent to the class `Converter`, the class is of course the receiver of the message. The phrase "... an instance of the *class Converter*..." would have also been correct. Among Smalltalk programmers, however, it's considered good style to avoid direct reference to class names as much as possible, because that would mean that the comment quoted above could not be changed should the class name be changed. It occasionally happens that the name of a class is changed during development, and the

development environment even supports a name change with a function for that purpose, which searches throughout the Image for all occurrences of the class name. But the search does not look in comments, and so the old name would remain there.

First of all, an instance of the class `Converter` must be created in the body of the method `withExchangeRate:..`. We could do that with the expression `Converter new`. But in this case, too, it would be better to avoid a direct reference to the class name and to write `self new` instead. We've already discussed the meaning of the pseudo-variable `self` in Sect. 6.2. At runtime, it takes the place of the object that received the message that precipitated the activation of the method. That means that in our case, `self` is bound to the class `Converter`.

The expression `self new` thus produces an instance of the class `Converter` when it is evaluated in a class method of that class. We also need to communicate to this instance that its rate of exchange is to be `aNumber`. That is, we must send this instance a message to that effect. And so we expand the expression with the line:

```
self new exchangeRate: aNumber
```

Since at this point we want to set the instance variable `exchangeRate`, we are following Smalltalk's naming conventions (see Sect. 6.2) and using the set method `exchangeRate:..`, which, however, still needs to be created as an instance variable.

Our class method `withExchangeRate` now looks like this:

```
withExchangeRate: aNumber
    "creates an instance of the receiver with the
     rate of exchange aNumber"
^self new exchangeRate: aNumber
```

Note that we've added the return operator to the beginning of the expression in the last line. This causes the newly created instance to be returned as an answer when it receives the message `withExchangeRate:..`. Forgetting this return operator is one of the most “popular” programming errors in Smalltalk. If it's missing, it's as though the last expression to be evaluated were `^self`. In our case, that would mean that the answer was the class `Converter` rather than the newly created instance.

Creating the Class Method in the System Browser

In order to add our class method `withExchangeRate:..` to the class `Converter`, it's first necessary for us to prepare by performing the following steps in the System Browser:

1. Highlight the class `Converter`.

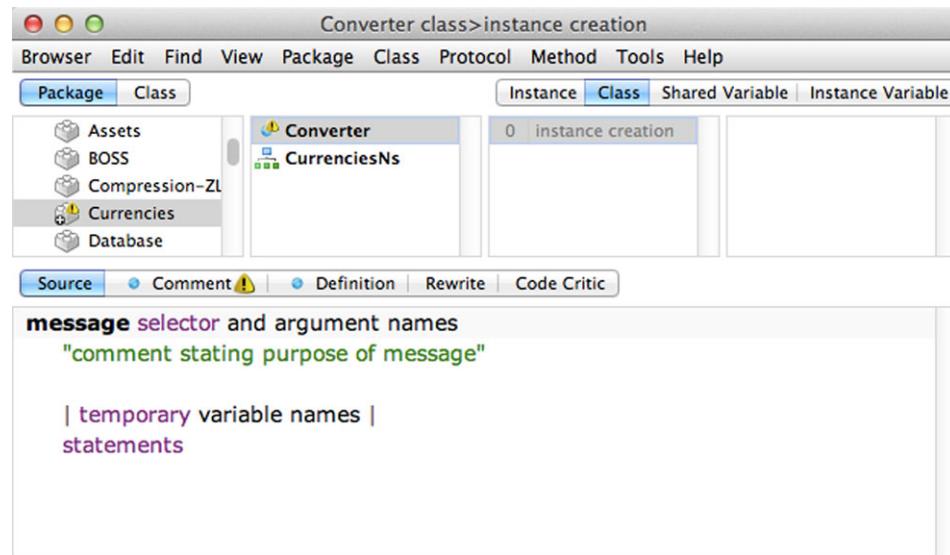


Fig. 7.7 Code template for new methods

2. Select the Class tab above Field 3.⁶
3. Select the method protocol `instance creation` (which the System Browser has already created, since instance creation is the main responsibility of class methods).

Once you've finished, the System Browser (see Fig. 7.7) displays in Field 5 a kind of template for writing new methods. The structure of our method `withExchangeRate:` resembles this template, except for the fact that we do not need any temporary variables.

We still have to perform the following steps in order to incorporate our method in the Image:

1. Replace the code template in Field 5 with the code for the method `withExchangeRate::`
2. The menu item **accept** (either from the **Edit** menu) must be activated. As long as the method text contains no syntax errors, this will translate the method into byte code. But the compiler will underline the message selector `exchangeRate:` with a dashed line (see Fig. 7.8) in order to advise the programmer about the missing `exchangeRate:` method. We have not yet defined a relevant set method.

Note that, once the method has been successfully accepted, the message selector appears in Field 4.

⁶See Fig. 5.14.

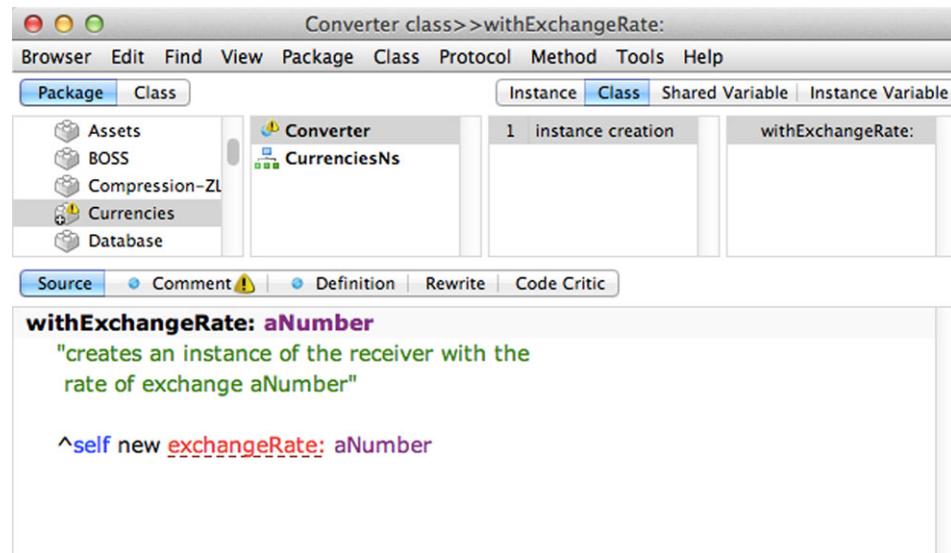


Fig. 7.8 The class method withExchangeRate:

But we still can't evaluate the expression `Converter withExchangeRate: 1.55` from our little test program, because the class method `withExchangeRate` uses the message `exchangeRate:` for which we must still define the matching set method as an instance method.

7.1.3 Defining Instance Methods

Among other things, creating the class `Converter` establishes, through the definition of its instance variables, the structure of the instances. What's still needed now is to give the instances of the class their desired behaviour. And this occurs by defining the appropriate instance methods. For example, there's the method for the message `convert` that we used in the test program. First, though, we'll define a method with the selector `exchangeRate:`, which we've already used within the class method `withExchangeRate::`. We're dealing with a method whose sole purpose is to set an instance variable (in this case `exchangeRate`), that is, a set method. This is only important at this point because Smalltalk uses the convention of collecting get and set methods together in a method protocol called `accessing`.

Creating an Instance Method in the System Browser

In preparation for adding our instance method `exchangeRate:` to the class `Converter`, perform the following steps in the System Browser:

1. Select the class `Converter`.

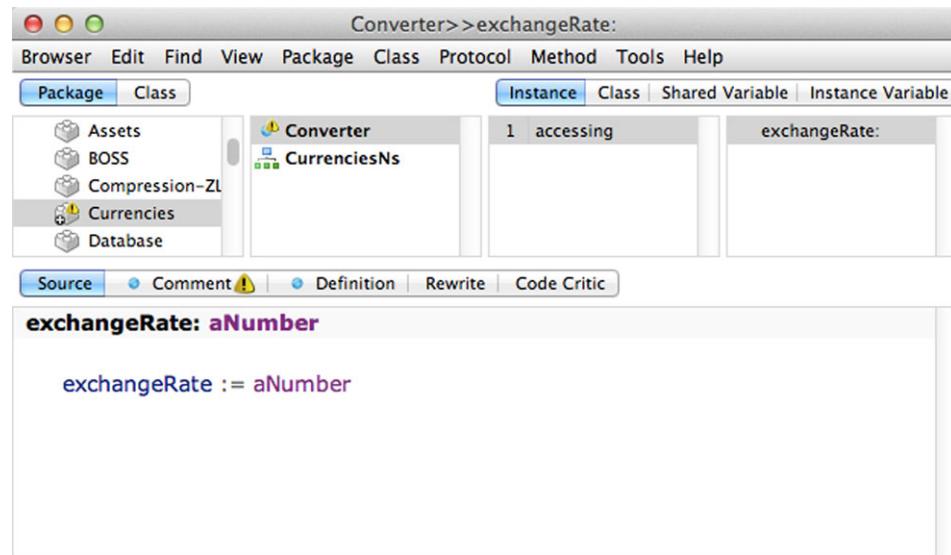


Fig. 7.9 The instance method `exchangeRate:`

2. Select the **Instance** tab above Field 3.
3. Select the method protocol `accessing` (in this case, automatically added by the System Browser, since nearly every class has get and set methods available).

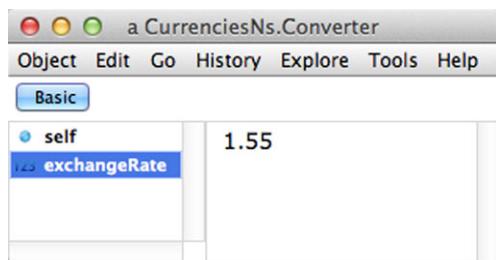
Once you've performed those steps, Field 5 shows a code template—as it did in Fig. 7.7—which we replace with the following method definition:

```
exchangeRate: aNumber
    exchangeRate := aNumber
```

In order to include this method in the Image, it must again be translated into byte code by selecting the menu item `accept` (either from the `Edit` menu or from the context menu for Field 5). If the method text is free of syntactic errors, this action displays the result shown in Fig. 7.9. Note:

- It's not customary in Smalltalk to put comments in the header of get and set methods, because their operation needs no further explanation.
- A return operator is not required in a set method. Returning the receiver (via the implicit `^self`) is the proper reaction of an object upon receipt of a set message.

Fig. 7.10 An initialized Converter object



Now we can evaluate the expression `Converter withExchangeRate: 1.55` from our little test program. When we use **Inspect it** to execute it in the workspace, the Inspector shows an initialised `Converter` object, that is, one with a meaningful initial value supplied for the instance variable (see Fig. 7.10).

In order to execute the test program

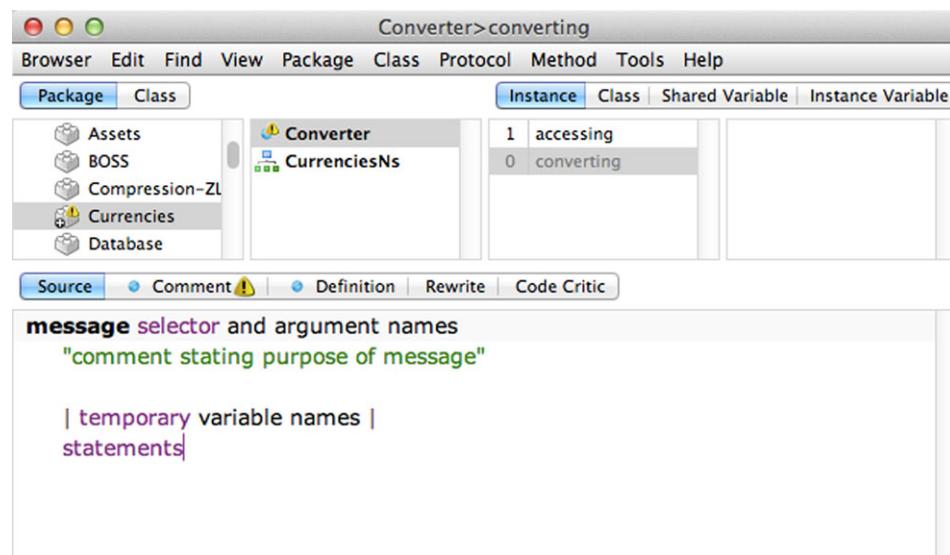
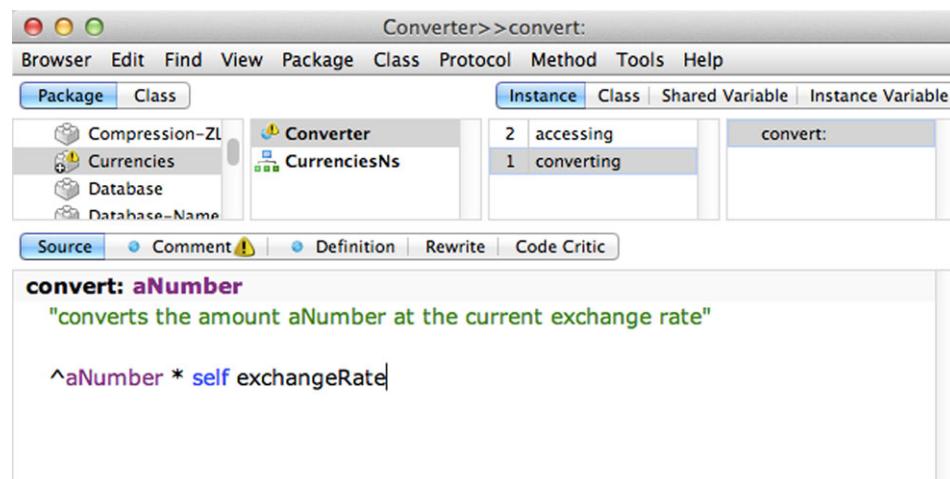
```
euroToDollar dollars |
euroToDollar := Converter withExchangeRate: 155.
dollars := euroToDollar convert: 227
```

we still need the following method definitions.

Definitions of the Methods `convert:` and `exchangeRate`

With `convert:` we are dealing again with an instance method, for which, however, we should create a new method protocol in the System Browser. That leaves the only protocol that our class `Converter` uses for instance methods, accessing, and it remains reserved for get and set methods. To create a new method protocol in the System Browser, with the class designator selected in Field 2, select the menu item **Protocol → New** or the corresponding command from the context menu in Field 3, then enter the name of the method protocol in the dialogue window. The System Browser will then look like the one shown in Fig. 7.11. The protocol name appears provisionally in italics and between parentheses to indicate that it does not yet contain any methods. At this point, we replace the method template that appears by default in Field 5 with the text:

```
convert: aNumber
    "converts the amount aNumber at the current exchange
     rate"
    ^aNumber * self exchangeRate
```

**Fig. 7.11** A new method protocol**Fig. 7.12** The instance method convert:

In order to access the instance variable `exchangeRate`, we use the get method with the same name in the expression `self exchangeRate`. Because we haven't yet defined the method, the message selector is again underlined by a dashed line. At this point, the System Browser looks as it does in Fig. 7.12.

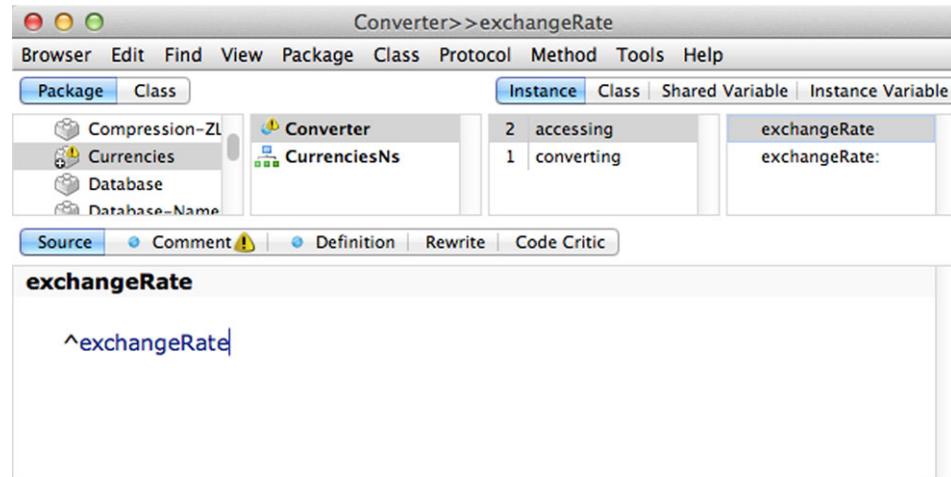


Fig. 7.13 The instance method `exchangeRate`

The get method `exchangeRate` still has to be created in the protocol `accessing`, as shown in Fig. 7.13. For the get method, it's important to remember to include the return operator.

Now we have defined all of the methods necessary to permit executing of the test program.

Executing the Test Program

When you enter this program in the workspace—supplemented by an output in

```
| euroToDollar dollars |
euroToDollar := Converter withExchangeRate: 1.55.
dollars := euroToDollar convert: 227.0.
Transcript show: '227 euros are ',
                dollars printString, ' Dollar'
```

Transcript—select it and execute it with **Do it**. The output should appear in the last line of the Transcript shown in Fig. 7.14.

7.1.4 Expanding the Converter

It would certainly be a reasonable thing to undertake a small expansion of the functionality of our `Converter` instances. A currency converter that can convert euros into dollars is also capable of performing the reverse calculation. To convert dollars into euros, all

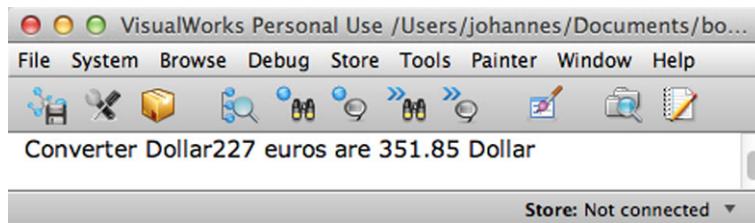


Fig. 7.14 Result of the test program

that one needs to do is divide the dollar amount by the value that's stored in the instance variable `exchangeRate`. For that purpose, we introduce a second conversion method:

```
convertInverse: aNumber
    "inversely converts the amount aNumber at the current
     exchange rate"

    ^aNumber / self exchangeRate
```

After we've created the method (once again in the protocol `convert`), we might use the following sequence to test it:

```
| euroToDollar euros |
euroToDollar := Converter withExchangeRate: 1.55.
euros := euroToDollar convertInverse: 351.85.
Transcript show: '351.85 dollars are ',
                  euros printString, ' Euro'
```

In the Transcript, we expect the text

```
351.85 dollars are 227.0 euros
```

to appear.

Afterthoughts

It must finally become clear that a program's legibility can be impaired by the fact that currency amounts are represented simply as numbers. It's difficult to know that the number 351.85 represents the amount \$351.85. In our test programs, this is helped along somewhat

by the use of meaningful variable names such as euros and dollars. Back toward the beginning of Sect. 7.1, we discussed the problem of representing currency amounts using floating-point numbers. There is thus much in favour of creating a special class, the instances of which are currency amounts and for which a commercially correct arithmetic could be used. At this point, though, we won't delve any deeper into this matter. A further solution to this problem that the class library of VisualWorks makes available is the use of the class `FixedPoint`, which we will make use of in the next section.

7.2 Case Study: Cinemas

In this section, we will analyse a (very) small requirements definition from the “real world” and lead it towards an object-oriented solution. The analysis of the requirements definition within the framework of object-oriented software development is called an *object-oriented analysis*, and the development of a suitable system of classes and objects is called *object-oriented design*. Texts about object-oriented software development usually handle in detail the processes used here and the principles that must be adhered to. As a representative sample of these texts, we'll mention Oestereich (2005) and Booch et al. (2007).

At this point, we'll suggest a drastically simplified process that uses the principle of text analysis to find objects and classes. We assume that the requirements for the program to be developed exist in the form of a text that's been formulated as precisely as possible—a prerequisite that is not often found in practice. In larger software projects especially, the requirements must first be mutually developed in an elaborate cooperation between the principals and the contractors.

Let's examine the following:

Problem Description

The owner of a suburban cinema can establish his own admission-ticket prices. He's determined empirically the relationship between the ticket price and the average number of attendees: At a ticket price of 5 €, an average of 120 people attend a showing. If he reduces the price by 0.10 €, another 10 people come. More attendees, though, increase his expenses. Each showing costs of 180 € plus 0.05 € for each viewer. The owner wants to know what his profit is at a particular ticket price.

7.2.1 Analysis of the Problem Description

We should start out by saying that the problem description doesn't inevitably suggest that one should solve it by developing an object-oriented program. Other programming styles⁷—such as *functional programming*—could be used just as well.

⁷Also called *programming paradigms*.

The last sentence of the problem description tells us that the owner wants to know the relationship between the ticket price and his profit. Beyond that, we can infer from the text the following relationships:

1. The profit is the difference between income and expenses. The text doesn't actually say that, but it's common knowledge. But the text at least contains no suggestion that the profit is to be calculated in some other way.
2. The income is the product of the ticket price and number of attendees.
3. The expenses are the sum of the fixed expenses (180 €) and the product of the number of attendees and the cost per attendee (0.05 €).

The text is not very precise with regard to the relationship between the number of attendees and the ticket price. It does not tell us

1. What happens to the number of attendees when the ticket price is lowered, nor
2. Whether the relationship is even linear.

We'd have to ask the principal to answer those questions. For simplicity's sake, we'll assume that the relationship between the number of attendees z and the ticket price p is shown by the formula

$$z = 120 + \frac{(5.0 - p) \cdot 15}{0.1} = 120 + (5.0 - p) \cdot 150$$

That is, we assume a linear relationship using the parameters given in the text. That means, for example, that the number of attendees decreases by 30 when the ticket price is raised by 0.20 €.

Finding Objects, Classes and Methods

A first attempt—which is all we'll do at the moment—at deriving suggestions for appropriate classes and methods from the problem description consists of a “grammatical” analysis of the text:

1. Noun phrases in the text refer to objects and classes or their properties.
2. Verbs and predicates, on the other hand, are more likely to represent processes and functions that describe the behaviour of objects and that might perhaps be implemented by methods.

The noun phrases in the text are: *owner, suburban cinema, ticket price, number of attendees, ticket, viewer, expenses, profit* and *showing*. We've eliminated noun phrases such as *average* and *relationship* that are not relevant to the problem, as well as synonyms (*ticket, admission ticket*). With regard to the further use of these noun phrases, we must answer the following questions:

1. Which noun phrases are actually relevant for solving the problem?
2. Do the noun phrases really characterise a relationship in an unambiguous way?
3. Which noun phrases characterise objects and which characterise object properties?

Question 1: The word *ticket* is surely meaningless for determining the profit, since what is important is the price that the viewer must pay. But that means that the *viewer* is also irrelevant. But this shows that the question can't actually be answered without also looking at the second part of the text analysis, concerning the functions to be realised. The two parts of the analysis can therefore not be performed entirely independently of one another.

Question 2: The word *expenses* is also ambiguous, because the text tells us that there are apparently variable and fixed expenses for each showing.

Question 3: Related to Smalltalk, one can say that the question is wrongly posed. That is, because properties in Smalltalk are, of course, also objects that are stored in the instance variables of objects, whose properties they describe. The word *properties* refers here to irreducible objects that contain no analysable structure. One might ask, for example, whether the ticket price should be regarded simply as a number or perhaps instead as an instance of a class for amounts of money.⁸

As a result of the **analysis of Part 1** of the text analysis, we have determined:

- The relevant noun phrases are *ticket price*, *number of attendees*, *expenses*, *profit* and *showing*. The phrase *suburban cinema* would be of interest only if we were able to convince the cinema owner (our principal) that he's certain to own more cinemas in the future and that he will want to determine separately the expected profit for each of the various cinemas.
- The ticket price, the expenses and the profit are attributes of a showing.
- The text contains a series of constants (for example, 120 attendees, 180 €). The text doesn't tell us how to handle them. We are assuming that the various numbers are intended only as examples. We want to keep our program flexible enough that these constants can be modified without needing to modify the program. After all, one can assume that expenses will change in the future. This leads to additional properties for a showing:
 - The base number of attendees (120)
 - The ticket price of the base number of attendees (5 €)
 - The attendee-price coefficient (15/0.10 €)

That means that the expense properties are

- Fixed expenses per showing (180 €)
- Cost per viewer (0.05 €)

⁸Compare the afterthoughts in Sect. 7.1.4.

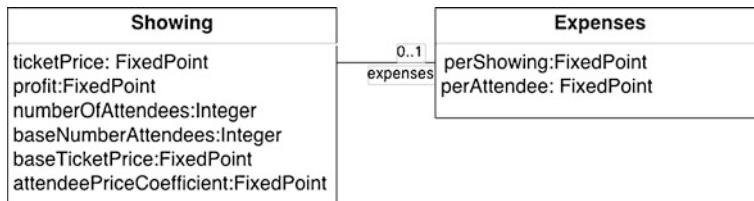


Fig. 7.15 Class diagram for the “cinema problem”

The (provisional) result of Part 1 of the text analysis can be summarised by the class diagram shown in Fig. 7.15. The following needs to be said about the diagram:

- The class `FixedPoint` is used as the attribute class for prices. Instances of this class are numbers with a defined precision and are thus by definition very suited for calculating amounts of money. Although VisualWorks contains this class, other Smalltalk libraries may be missing it.
- It is not really necessary to keep the *expenses* together in their own class. In this case, it serves to make the application modular. That means that it's easier to replace the “expense module” with another module should the determination of expenses occur differently in the future than it would be if the expense determination were integrated into the class `Showing`. This illustrates the principle of *separation of concerns*. A well-defined, discrete partial task is placed into its own module (its own class).

Part 2 of the text analysis: There are basically three activities described in the text of the problem description:

- The main activity (apparently the only one that interests the principal) is the determination of his profit as a factor of the ticket price.
- As part of that determination, we must also determine the number of attendees based on the empirical formula, again as a factor of the ticket price.
- Finally, income and expenses—both of them dependent on the ticket price and the number of attendees—must be determined; the difference between them is the profit.

At this point, we notice that the properties `numberOfAttendees` and `profit` are unnecessary in the class `Showing`, since both values can be determined from the other values. For that reason, Fig. 7.16 shows a class diagram reduced by those properties.

7.2.2 Implementation

The implementation presented here is limited to affecting the cinema owner's requirements exactly as prescribed by the problem description. The only parameter that can be changed—or entered—is the ticket price, which can be supplied during the creation of a

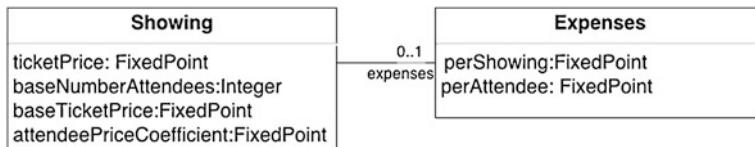


Fig. 7.16 Simplified class diagram for the “cinema problem”

Showing instance. The only calculation that can be carried out is determining the profit. In the workspace, that might look like this:

```
(Showing withTicketPrice: 5.00s) profit
```

This means that an instance of the class **Showing** with a ticket price of 5.00 € is created, to which the message `profit` is subsequently sent. As long as our methods were correctly implemented, the evaluation of this expression must yield 414.00s, which one can easily verify with a pocket calculator.

► **Note:** Literals of the class `FixedPoint` are written with a lowercase “s” appended at the end. The number of digits after the decimal point indicate the absolute accuracy of the constants. The entry `5.00s` thus indicates a fixed-point number with an accuracy to the hundredths place.

Creating Instances of the Class **Showing**

Figure 7.17 shows the definition of the class **Showing** in the System Browser. The instance variables represent the properties from Fig. 7.16. We can already see that the class method `withTicketPrice:` has been placed in the protocol `instance creation`. In addition, the figure shows that there is a package `Cinemas` and a namespace `CinemaNs`.

The implementation of the class method `withTicketPrice:`:

```

withTicketPrice: aFixedPoint
    "created an instance with ticket price aFixedpoint"

    ^self new initialize ticketPrice: aFixedPoint

```

first creates an instance of the class, to which are then sent the messages `initialize` and `ticketprice: aFixedPoint`.

The implementation of the instance method `initialize`

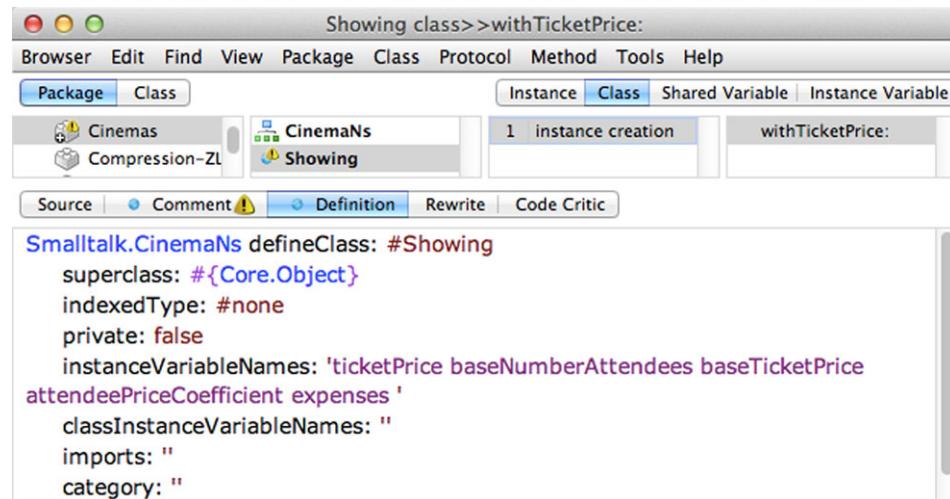


Fig. 7.17 Class definition Showing

```

initialize
self
baseNumberAttendees: 120;
baseTicketPrice: 5.00s;
attendeePriceCoefficient: 150.00s;
expenses: Expenses new

```

uses the set methods of the instance variables in order to populate them with the constants described in the problem description. An instance of the class `Expenses` is stored in the instance variable `expenses`. When the instance is created, it will also be populated with the constants for fixed expenses and the cost per attendee supplied in the problem description (see below).

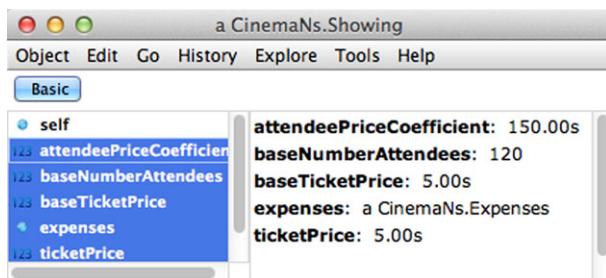
Assuming that all set methods and the class `Expenses` (see Fig. 7.19) have been created, when you use **Inspect it** to evaluate the expression

```
Showing withTicketPrice: 5.00s
```

it yields the instance of the class `Showing` that is shown in Fig. 7.18.

Calculating the Number of Attendees and the Profit

The next section starts with the implementation of the method `profit`. We will use top-down design or stepwise refinement.

Fig. 7.18 A showing

The following implementation of the instance method `profit`

```
profit
  ^(self incomeAt: self numberOfWorkers)
  - (self expensesAt: self numberOfWorkers)
```

calculates the profit based on the income and expenses, for the calculation of both of which the number of workers is required. That number is calculated in the method `numberOfWorkers` (using the formula that the cinema owner arrived at empirically). Precisely how that's done is not important. Nevertheless, the number of visitors is unnecessarily calculated twice, which the following alternative implementation avoids.

```
profit
  | numberOfWorkers |
  numberOfWorkers := self numberOfWorkers.
  ^(self incomeAt: numberOfWorkers)
  - (self expensesAt: numberOfWorkers)
```

The implementation of the instance methods `incomeAt:` and `expensesAt:` are now very easy:

```
incomeAt: numberOfWorkers
  ^numberOfWorkers * self ticketPrice.

expensesAt: numberOfWorkers
  ^self expenses expensesAt: numberOfWorkers
```

Since it was decided to “exile” the calculation of the expenses of a showing to the class `Expenses`, the method `expensesAt:` delegates the calculation to the `Expenses` object that's stored in the instance variable `expenses`.

The implementation of the instance method `numberOfAttendees`

```
numberOfAttendees
  ^ (self baseNumberAttendees)
    + (self attendeePriceCoefficient
      * (self baseTicketPrice - self ticketPrice))
  rounded
```

represents the implementation of the formula that the cinema owner arrived at empirically. The message `rounded` is sent to the result of the formula, which rounds the calculated number of attendees so that it appears as a whole number. This is, however, not absolutely necessary. Because

1. Instead of rounding, one could use the message `truncated` to truncate the number, which would cause any further calculations to be made with the next lower number.
2. Or one could also continue to calculate with a non-whole number, since the calculation of the profit is based on experiential and average values anyway.

Notice that the development of the methods used in the profit calculation can occur without knowledge of the details of the expense calculation, which is hidden within the class `Expenses`.

The Class Expenses

Figure 7.19 shows that—alongside the definition of the class `Expenses`, which follows the class diagram shown in Fig. 7.16—a new method also exists in the protocol `instance creation`.

The implementation of the class method `new`

```
new
  ^super new initialize
```

anticipates simply initialising the `new` object as well as creating an instance.

► **Note:** The expression `super new` causes the creation of an uninitialised object of the class `Expenses`. Using the pseudo-variable `super` instead of `self` causes the search for the method `new` to begin in the superclass of `Expenses`, that is, in `Object`. You can find detailed instructions for the use of the pseudo-variables `self` and `super` in Sect. 11.2.

The implementation of the instance method `initialize`

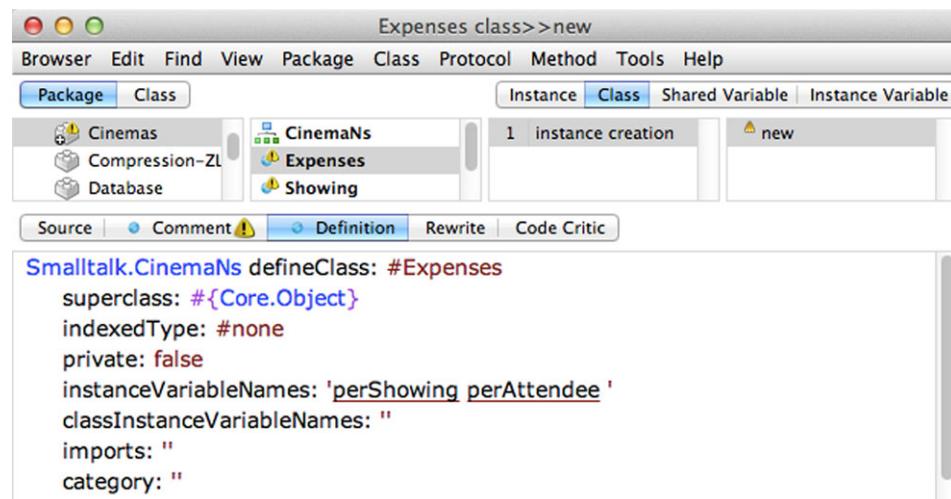


Fig. 7.19 Class definition Expenses

```
initialize
self
    perShowing: 180.00s;
    perAttendee: 0.05s
```

uses the corresponding set methods to populate the instance variables with the constants from the problem description. Beyond that, only the implementation of the instance method `expensesAt`:

```
expensesAt: numberOfAttendees
^self perShowing
+ (numberOfAttendees * self perAttendee)
```

is significant.⁹

Checking the Implementation

This concludes the implementation of the two classes and their methods. If everything was correctly programmed, the evaluation of the expression

⁹As in the class `Showing`, we skip the specification of the get and set methods.

```
(Showing withTicketPrice: 5.00s) profit
```

yields the value 414.00s.

For additional testing, one can use the following expressions:

- (Showing withTicketPrice: 4.90s) profit yields 474.75s
- (Showing withTicketPrice: 5.10s) profit yields 350.25s
- (Showing withTicketPrice: 4.00s) profit yields 886.50s

At the end of this section, we'll remark again that, obviously, all aspects of the design of object-oriented programs could hardly be dealt with on the basis of this little case study. This book doesn't intend to do that, though. An interested reader can find extensive literature for further study of this topic.

7.3 Defining Class Variables

For the sake of completeness, at this point, we'll discuss how class variables can be defined, even though there is actually no meaningful application for them in the sample classes examined in this chapter. In VisualWorks, class variables are viewed as a special category of non-private or shared variables. A class variable is a shared variable of a class and all of its instances. They can be accessed from both the class methods and the instance methods of the class. The class and all of its instances "see" the same object when they access a class variable.

Class variables are inherited by the subclasses of the class within which they were defined, and are thus also accessible to the instances of these subclasses. In contrast, objects from other "foreign" classes are not able to access them.

You can see a meaningful use of class variables in the class `Date`, whose instances represent date information: A class variable `MonthNames` points to an array that contains symbols for the various month names. It would certainly be a waste of storage space if one wanted to store information about the names of the months in every date object. Section 8.1.9 contains other meaningful uses of class variables.

We'll limit ourselves here to a description of how to create new class variables. In VisualWorks, perform the following steps:

1. First, select the class in the System Browser for which a class variable is to be defined, then select the Shared Variable tab.
2. In Field 3 of the System Browser, add a category, for example, `class variables`. The browser will show a screen like the one in Fig. 7.20.
3. In the first line of the template for the definition of a class variable that now appears in Field 5 of the System Browser, replace the parameter of the keyword `defineSharedVariable:` with the symbol of the class variable you are creating,

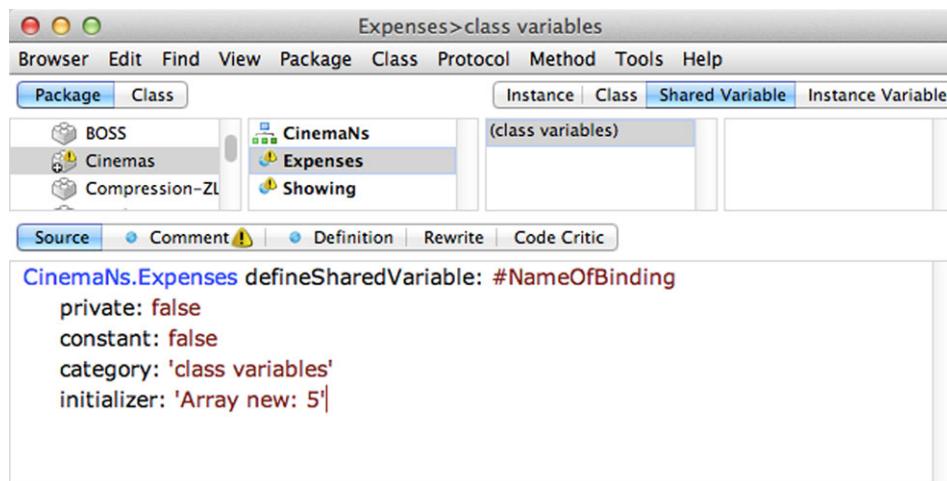


Fig. 7.20 Template for the definition of a class variable in the System Browser

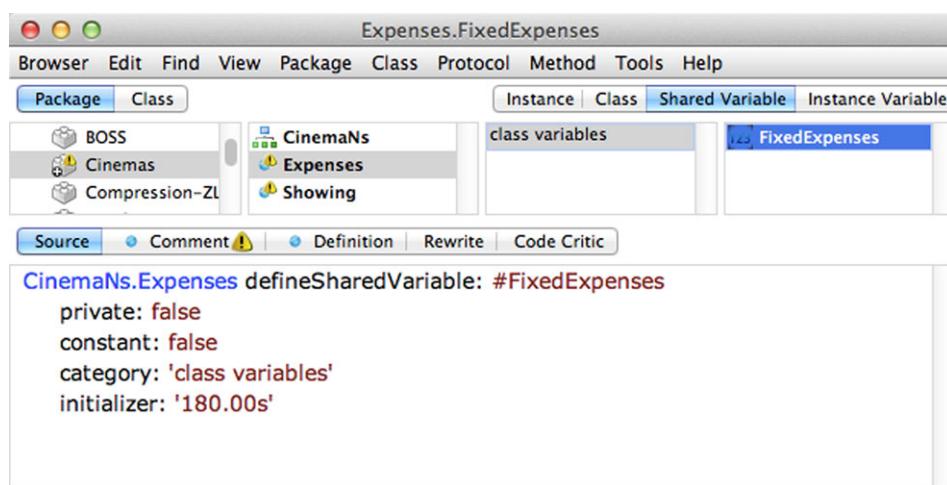


Fig. 7.21 Definition of the class variable FixedExpenses

for example, `#FixedExpenses`. In Smalltalk practice, the names of class variables always start with an uppercase letter.

4. The parameters of the keywords `private:`, `constant:` and `category:` remain unchanged.
5. You can specify a character string containing a Smalltalk expression for the parameter of the keyword `initializer:`. In that case, the class variable will be initialised with the result object of the evaluation of the expression. If you do not want to initialise the variable, replace the character string with '`'nil'`'.

6. Finally, click **Accept** on the context menu of the System Browser to create the class variable.

The result of the definition of the class variable `FixedExpenses` is shown in Fig. 7.21. The initialisation of the variable with the value 180.00s occurs after you select the menu entry **Method → Shared Variable → Initialize** in the System Browser.

A class variable `FixedExpenses` might make sense in the cinema application in Sect. 7.2 if there were many showings and therefore many `Expenses` objects. But if the fixed expenses always remained the same, it would not make sense to store them in an instance variable in every single `Expenses` object.

The size of Smalltalk as a programming language is really very small when compared to other object-oriented languages. This certainly appears to be the case when you measure its size on the basis of the means of expression that the language's syntax puts at the programmer's disposal. There are basically only three types of messages and just a few reserved words, such as `nil`, `self`, `true` and `false`. The language really becomes a powerful tool for the programmer when it's used with the class library, that is, with the mass of programming code that already exists and which can be used in appropriate ways for your own purposes.

The class library in the version of VisualWorks that was used in the creation of this book contains more than 1700 classes. Within this extensive library, lie both the power of the development environment and the programmer's difficulty at navigating that environment. It's scarcely possible for a single individual to know all the classes, but the application development doesn't require the programmer to do so, especially since—except in the source code—they're not completely documented anywhere.

Of great assistance in learning to program in Smalltalk is the study of existing classes and their methods, which are present to a great extent in the source code. The following sections will help you orient yourself by discussing a selection of the most important classes for application development.

This chapter also examines in greater detail two essential concepts of object-oriented programming, *inheritance* and *polymorphism*. The two concepts are bound together in a close relationship with the concept of classes and of class hierarchy. One must become totally familiar with these concepts in order to write good *object-oriented* programs.

As an aid in this process, Sect. 8.5 takes up the problem introduced in Sect. 2.3, "Solving a Quadratic Equation". Back then, we looked at the problem from the point of view of an algorithm, while in this chapter we consider what an *object-oriented* solution would look like, making use of class hierarchies, inheritance and polymorphism.

8.1 The Smalltalk Class Hierarchy

8.1.1 Structure

The basic structure of a Smalltalk class library is determined by the following fundamental rules:

1. The class `Object` is the common superclass of all classes. It therefore represents the root of the class tree. It has no superclass.
2. Each class other than `Object` has exactly one superclass. So-called multiple inheritance, where one class can have several superclasses, does not exist in Smalltalk.

The library contains a series of classes that are of no importance to a “normal” programmer. These include:

- Classes that are needed for the implementation of the development environment itself, that is, for the inner workings of VisualWorks. The various components that we’ve gotten to know (Browser, Inspector, Debugger, etc.) are implemented in Smalltalk.
- So-called metaclasses, classes of classes. These are necessary because in Smalltalk, classes are also objects and thus instances of a class, that is, of their respective metaclass. Nevertheless, the development environment is designed in a way that the programmer is largely unaware of these metaclasses. We will discuss this further in Sect. 11.3.
- Classes that form the interface with the VM. These include especially those that are part of the implementation of the compiler that translates Smalltalk methods into byte-code that the VM can interpret.

Figure 8.1 shows a portion of the Smalltalk class library that includes classes that are especially important for application development. Among them are:

- Number classes (`Number` and its subclasses)
- Date and Time classes used to handle date and time specifications
- Boolean, True and False classes for truth values
- Container classes (`Collection` and its subclasses)
- Character (`Character`) and Character String (`String` and `Symbol`) classes

We must emphasise that the class diagram in Fig. 8.1 obviously shows only a tiny section of the entire class hierarchy. Furthermore, it does not show all of the classes that application programmers need to concern themselves with. For example, the development of modern interactive programs also needs the classes that permit the realisation of graphical user interfaces. One example for the use of these classes is the development environment of VisualWorks itself. But since this book does not deal with the programming of graphical user interfaces, we will not look more closely at the relevant classes.

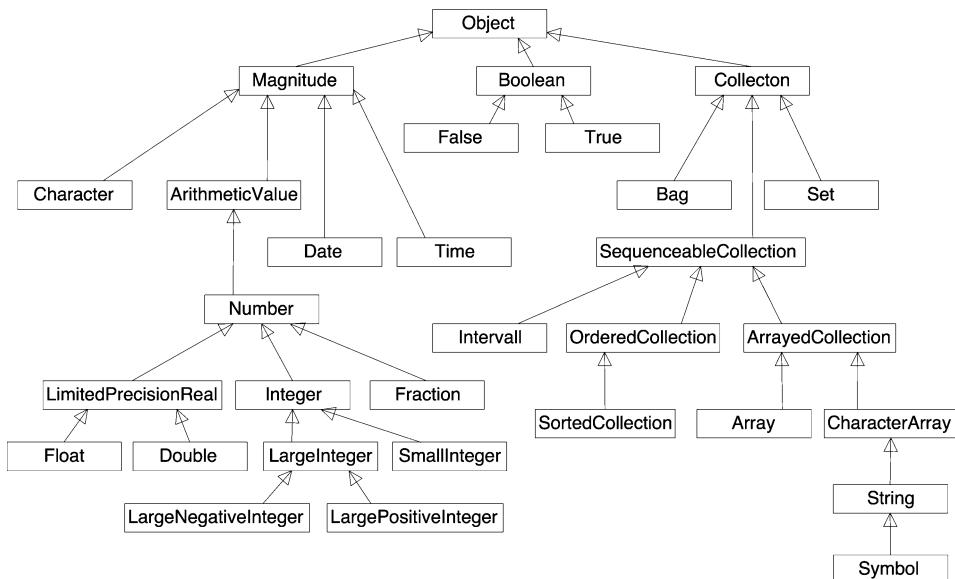


Fig. 8.1 A section of the Smalltalk class hierarchy

In this chapter, we will cover the classes for elementary objects like numbers (Sect. 8.1.2), truth values (Sect. 8.1.7), characters and character strings (Sect. 8.1.8) and date and time specifications (Sect. 8.1.9). Chapter 10 deals with Collection classes.

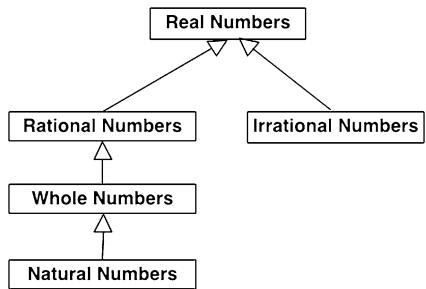
8.1.2 Smalltalk’s Number Concept

From a mathematical viewpoint, the various kinds of numbers are defined by an infinitely large quantity of numbers with individually identifying characteristics. This reveals an initial difficulty: By definition, infinitely large quantities cannot be represented in a finite machine like a computer. For technical reasons, certain limitations are thus necessary on the machine representation of numbers in comparison to their mathematical-theoretical counterparts. The various programming languages differ markedly in how they manage this problem.

Leaving complex numbers aside for the moment, from a mathematical point of view, number types can be classified according to the “class hierarchy” shown in Fig. 8.2. In this case, the relationship between subclasses and superclasses is always to be interpreted in the sense of “is a”. A natural number *is a* number. A whole number is *a* rational number, etc.

In the next sections, we will examine how the corresponding Smalltalk classes accomplish the technical realisation of the mathematical types of numbers in a bottom-to-top hierarchy.

Fig. 8.2 Hierarchy of Mathematical number types



Natural and Whole Numbers

Very few programming languages have an independent way to technically represent natural numbers. That's true for Smalltalk too. In a mathematically correct way, natural numbers are regarded as a subset of whole numbers, which are represented by the class `Integer` and its subclasses. There are also technical reasons for why there are five classes for whole numbers (see Fig. 8.1) rather than just a single one.

The processor for the real machine—on which, of course, the VM is running as a program—has at its disposal hardware-based commands for the four basic arithmetic calculations, and especially for whole numbers. The size of these whole numbers is limited by the length (measured in bits) of a so-called machine word. The word length of today's computers is frequently either 32- or 64-bit. In other words, for a word length of 32 bits, the processor hardware can only process whole numbers that can be represented in 32 bits, where 31 bits are used for the magnitude and 1 bit for the sign. That means that the largest positive whole number that can be displayed in this way is

$$2^{31} - 1 = 2147483647$$

For negative whole numbers, there is a corresponding limitation downwards.

When the result of an arithmetic operation exceeds the number range that can be represented, using processor hardware to realise whole numbers, the hardware sets signal bits to indicate this. It then becomes the responsibility of the software to react in an appropriate way. Such an occurrence is referred to as an *arithmetic overrun*.

Many programming languages are content to represent whole numbers as a data type usually referred to as an `integer` that corresponds exactly to the machine's own representation. In other words, there is a smallest and a largest integer, whose magnitudes depend on the processor hardware. In that case, an arithmetic overrun means that the result can no longer be represented as an integer. Depending on the programming language or the compiler, this can cause the program either to abort or—what's worse—continue to calculate but with an incorrect result. This shows that it's ultimately the programmer's responsibility to prevent arithmetic overruns. They must be especially vigilant when programming complex arithmetic expressions that no intermediate results occur that are either too large or too small.

For simplicity's sake, let's assume that the largest possible integer is 100. In that case, the expression

$$60 + (50 + (-40)) = 60 + 10 = 70$$

can be evaluated without a problem, while an expression with what is mathematically the same value

$$(60 + 50) + (-40)$$

is undefined, because $60 + 50$ yields an intermediate result that is too large. This very unsatisfactory behaviour of integer arithmetic is ameliorated only by the fact that the magnitudes for the largest and smallest integers are in reality considerably larger than in our example.

The Class `Integer` and Its Subclasses

Smalltalk handles this problem differently. The basic rule is that the magnitude of a whole number can be any size as long as the memory has enough space available to store the digits. If that isn't the case, though, then the real machine is in any event not suited to executing the program.

As long as a whole number does not leave the representable range determined by the word-length of the processor hardware, it is allocated to the class `SmallInteger`. That allows the VM to efficiently use the processor hardware to directly process numbers that are of "small" magnitude.

The VM uses software to calculate larger numbers. To that end, it must be capable of recognising the hardware's overrun signal. Numbers that lie outside the range of the `SmallInteger` class are automatically assigned to the classes `LargePositiveInteger` or `LargeNegativeInteger`. Although it costs much more processing time to process `LargeInteger` numbers than `SmallInteger` numbers, that's a reasonable trade-off compared to the alternative of not being able to process such numbers at all and having to experience a program abort. Furthermore, the programmer doesn't need to worry about which subclass of `Integer` an individual whole number falls into; the VM takes care of that automatically. Although one doesn't need the information as a general rule, one can query SmaViM for the values of the smallest and largest `SmallInteger` numbers by sending the messages `minVal` and `maxVal` to the class `SmallInteger`. In VisualWorks, for example, the evaluation of the expression

```
SmallInteger maxVal
```

yields the value 536870911. Adding 1 to that value results in an instance of the class `LargePositiveInteger` (see Fig. 8.3).

Section 8.1.3 describes further details of the `Integer` classes.

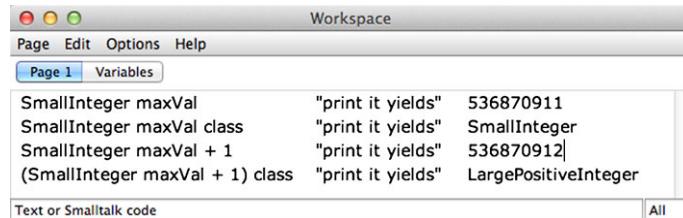


Fig. 8.3 Automatic assignment of whole numbers to the appropriate Integer classes

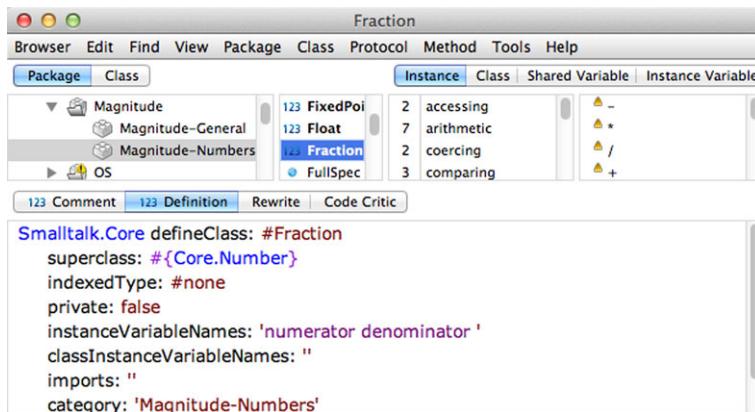


Fig. 8.4 Definition of the class Fraction

Rational Numbers

Rational numbers can always be written as a fraction consisting of a whole-number numerator and a whole-number denominator. Whole numbers are a special kind of rational number, the denominator of which is always 1. Figure 8.2 indicates this fact by showing that the class of whole numbers is a subclass of rational numbers.

Smalltalk provides the class `Fraction` for work with fractions. Objects of this class have two instance variables, `numerator` and `denominator` (see Fig. 8.4). And the Numerator and Denominator are instances of the Integer classes.

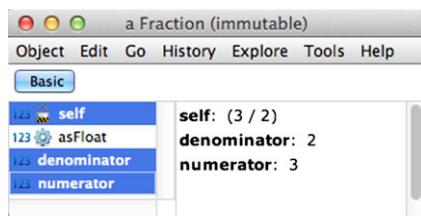
Fraction numbers have their own literal representation. Evaluating the expression

```
3 / 4
```

yields the object Fraction

```
(3/4)
```

Fig. 8.5 Fractions are always reduced: the result of $(6/8) + (3/4)$



The VM always reduces fractions as much as possible. One can illustrate this by evaluating the expression

```
((6/8) + (3/4)) inspect
```

The resulting Fraction object, shown in Fig. 8.5, is reduced to show numerator = 3 and denominator = 2.

In the Smalltalk class hierarchy (see Fig. 8.1), the class Integer is not placed—as one would expect from its mathematical classification—as a subclass of Fraction; rather it sits on the same level as Fraction. There are technical reasons for this deviation from the mathematical viewpoint. For if one placed Integer as a subclass of Fraction, then structural inheritance would dictate that each Integer object would also have the two instance variables numerator and denominator, and the latter would always have the value 1. That would mean that two Integer objects would have to be stored in memory for each whole number, which would be an enormous waste of memory.

This example shows that technical considerations sometimes prevent a conceptual hierarchy arising from the application context from being translated into a structurally identical class hierarchy.

Section 8.1.4 describes further details of the class Fraction.

Real Numbers

From a mathematical viewpoint, real numbers include both rational and irrational numbers. The latter are defined as numbers that cannot be represented as terminating or repeating decimals. That means, though, that as decimals, they can never be written exactly, but only approximately. Examples of irrational numbers include $\sqrt{2}$, π , or e . Because computers can only store a limited number of digits, the real numbers that they implement can only ever be a more or less coarse approximation of their desired mathematical concept. Approximations of real numbers are usually represented in a computer as so-called floating point numbers.

Since real numbers can only ever be represented approximately on a computer, all arithmetic operations are burdened with a rounding error. For that reason, one must work espe-

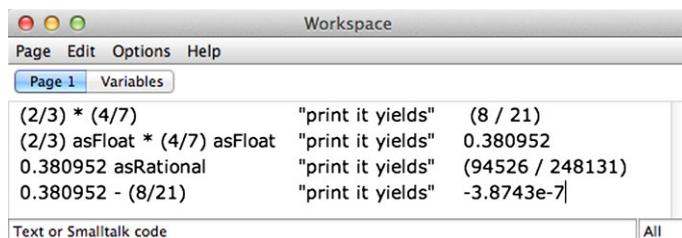


Fig. 8.6 Calculating with floating point numbers is inexact

cially carefully with the results of operations involving floating point numbers. For example, it usually makes no sense to test them for equality.

Besides fractions, floating point numbers are therefore a second means of representing non-whole numbers. While `Fraction` objects, though, are always exact, floating point numbers may sometimes evince a rounding error. The number examples shown in Fig. 8.6 make this clear. The message `asFloat` transforms a fraction into a floating point number, while the message `asRational` transforms a floating point number into a fraction. Since the floating point number 0.380952 is not exactly equal to $8/21$, the difference between them, -3.8743×10^{-7} ($= -3.874 \cdot 10^{-7}$), may represent a very small number, but it is not 0.0.

In the modern Smalltalk class library, floating point numbers are represented with the two classes `Float` and `Double`, both of which share the superclass `LimitedPrecisionReal` (see Fig. 8.1). Based on the above considerations, it is not reasonable to regard the class `LimitedPrecisionReal` as the Smalltalk equivalent of real numbers and to order the other number classes below it.

Section 8.1.4 describes further details of the classes `Float` and `Double`.

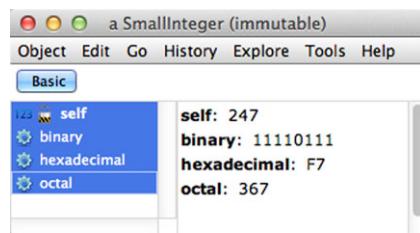
8.1.3 The Integer Classes

Literals

Whole numbers are written as a simple series of decimal numerals with or without a leading sign; in all cases, though, they are written without a decimal point. (Numeral strings containing a decimal point are instances of the classes `Float` or `Double`.) The general assumption is that the numbers are written in base-10, that is, that the numerals right to left are ones, tens, hundreds, etc. In Smalltalk though, it's also possible to use whole-number literals in other bases. In that case, the base, followed by the lowercase letter `r`, is written before the actual numeric string. Thus, for example,

- 2r1111011 is base-2
- 8r367 is base-8
- 16rF7 is hexadecimal

Fig. 8.7 A number represented in various base systems



All three have the same decimal value, $10r247$. Of course, you can omit the sign $10r$ for base-10 numbers. One can assure oneself however that the numbers are equivalent by examining the number 247 with the Inspector (see Fig. 8.7).

Any positive whole number greater than 1 can be used as a base. If the base is greater than 10, letters are used in alphabetical order for numbers greater than 9.

Methods

Figure 8.8 shows the protocols for the class `Integer`. Some protocols and their methods are of significance only for matters of implementation; as a rule they don't matter much to application programmers. For that reason, we'll only examine a few selected examples.

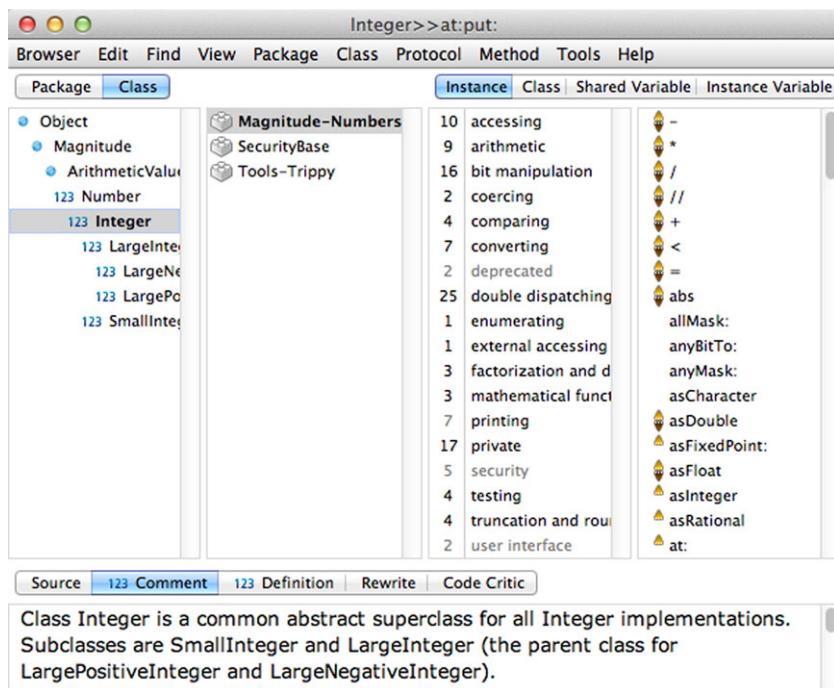


Fig. 8.8 The instance method protocols for the class `Integer`

Table 8.1 Arithmetic operations for `Integer` objects

Message pattern	Meaning
<code>+ aNumber</code>	Produces the sum of the receiver and the argument <code>aNumber</code>
<code>- aNumber</code>	Produces the difference between the receiver and the argument <code>aNumber</code>
<code>* aNumber</code>	Produces the product of the receiver and the argument <code>aNumber</code>
<code>/ aNumber</code>	Produces the quotient of the receiver and the argument <code>aNumber</code>
<code>// aNumber</code>	Produces the whole-number quotient of the receiver and the argument <code>aNumber</code> . The quotient is trimmed in the direction $-\infty$
<code>\\" aNumber</code>	Produces the remainder of the whole-number quotient of the receiver and the argument <code>aNumber</code> . The quotient is trimmed in the direction $-\infty$
<code>abs</code>	Produces the absolute value of the receiver
<code>negated</code>	Produces the negation (leading sign change) of the receiver
<code>quo: aNumber</code>	Produces the whole-number quotient of the receiver and the argument <code>aNumber</code> . The quotient is trimmed in the direction 0
<code>rem: aNumber</code>	Produces the remainder of the whole-number quotient of the receiver and the argument <code>aNumber</code> . The quotient is trimmed in the direction 0
<code>reciprocal</code>	Produces the reciprocal of the receiver, which may not be 0

Table 8.1 shows an overview of the arithmetic operations for `Integer` objects as receivers. The table also contains the operations `\\"`, `rem:` and `reciprocal`, which do not appear in the protocol `arithmetic` for the class `Integer` (see Fig. 8.8). These operations are defined in the class `Number` and have been inherited and can thus be used on `Integer` objects as well.

The Four Basic Arithmetic Functions

The first four operations in the table represent the “normal” four basic arithmetic functions and need no other explanation. With regard to the class membership of the result object, though, a few explanations are in order. It is true for all operations that require an argument that the argument may be an object of any number class. That means that an expression in the form `1 + 1.0` is also permissible. But the question remains: Is the result of this operation an `Integer` object or a `LimitedPrecisionReal` object?

For addition, subtraction and multiplication with an `Integer` object as the receiver, the answer is the same in all cases: If the class membership of the argument `aNumber` is

- an `Integer`, then the result is an `Integer` object.
- a `LimitedPrecisionReal`, the result is a `LimitedPrecisionReal` object.
- a `Fraction`, then the result is a `Fraction` object if the result does not allow itself to be reduced to a whole number; otherwise it is an `Integer` object.

These rules basically also apply to division. The argument is an `Integer` object; if the remainder of the division is not equal to 0, the result will be a `Fraction` object.

Fig. 8.9 Examples of arithmetic operations with Integer receivers

The screenshot shows a Smalltalk workspace window titled "Workspace". The menu bar includes "Page", "Edit", "Options", and "Help". The "Page" menu is currently selected. The "Variables" tab is also selected. In the main pane, there is a table-like list of operations and their results:

4 * 23	"print it yields"	92
2 + 2.0	"print it yields"	4.0
3 - (1/3)	"print it yields"	(8 / 3)
17 / 51	"print it yields"	(1 / 3)
51 / 17	"print it yields"	3
2 / (2/3)	"print it yields"	3
2 / 3.0	"print it yields"	0.6666667
4 / 2.0	"print it yields"	2.0

Below the table, there are two buttons: "Text or Smalltalk code" and "All".

The application of these rules is shown in Fig. 8.9.

Whole Number Division

The operations `/` and `quo:` produce the quotients of whole-number division; the operations `\` and `rem:` produce the remainders of the quotients. The result of whole-number division is always a whole number. It is, one could say, the answer to the question: How many times is the divisor (the message's argument) contained in the dividend (the receiver of the message), and what is the remainder? That means that 7 divided as a whole number by 3 yields 2 as a quotient and 1 as a remainder. This clear explanation fails, unfortunately, when one or both of the operands in the division have a negative leading sign. The question, how many times the number -3 is contained in the number 7 appears perverse. At the same time, one could be arbitrary and determine mathematically the result of the question, what 7 divided as a whole number by -3 should be. In mathematics, though, there are two different definitions, which leads to the answer that in Smalltalk (as well as in other programming languages) two pairs of operations exist for creating the quotients and the remainders. In a case where both operands are positive or both negative, the two operation pairs behave in the same way.

Common to both definitions is the fact that the following equation must be valid:

$$\text{Dividend} = \text{Quotient} \cdot \text{Divisor} + \text{Remainder}$$

For the example $\text{Dividend} = 7$ and $\text{Divisor} = -3$, this equation can be satisfied with the results:

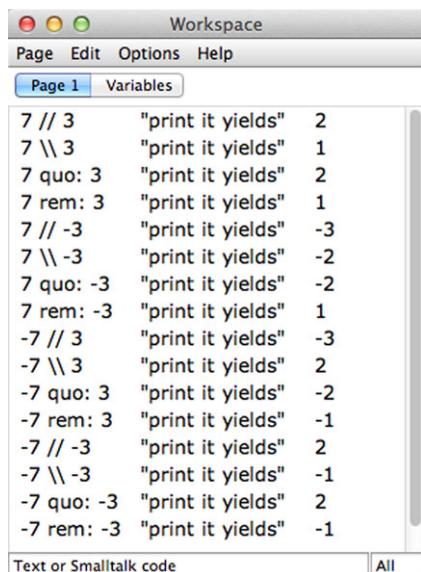
$$\text{Quotient} = -2 \quad \text{and} \quad \text{Remainder} = 1$$

and

$$\text{Quotient} = -3 \quad \text{and} \quad \text{Remainder} = -2$$

The first variation follows the rule that's occasionally met with in mathematics to the effect that the remainder of a division process is always positive. This rule is realised in Smalltalk

Fig. 8.10 Quotients and remainders of whole-number division



The screenshot shows a Smalltalk workspace window titled "Workspace". The menu bar includes "Page", "Edit", "Options", and "Help". A toolbar with three colored circles (red, yellow, green) is at the top. The main area displays a list of division operations and their results:

7 // 3	"print it yields"	2
7 \\\ 3	"print it yields"	1
7 quo: 3	"print it yields"	2
7 rem: 3	"print it yields"	1
7 // -3	"print it yields"	-3
7 \\\ -3	"print it yields"	-2
7 quo: -3	"print it yields"	-2
7 rem: -3	"print it yields"	1
-7 // 3	"print it yields"	-3
-7 \\\ 3	"print it yields"	2
-7 quo: 3	"print it yields"	-2
-7 rem: 3	"print it yields"	-1
-7 // -3	"print it yields"	2
-7 \\\ -3	"print it yields"	-1
-7 quo: -3	"print it yields"	2
-7 rem: -3	"print it yields"	-1

Below the list, there are two buttons: "Text or Smalltalk code" and "All".

using the operation pair `quo:` and `rem:`; the second variation results from the operation pair `//` and `\\\`.

If one examines the non-whole-number quotient

$$7/-3 = -2.33333$$

it becomes clear that the operation `//` for the whole-number quotient takes the next whole number in the direction $-\infty$, that is, -3 , while the operation `quo:` truncates in the direction 0, as explained in Table 8.1. Figure 8.10 shows additional examples of whole-number division. The application of whole-number division on non-whole-number divisors is not shown, because as a rule this rarely occurs in a practical application.

Mathematical Functions

A variety of mathematical functions can be used on `Integer` objects; most of them, though, apply to all types of numbers. These functions are discussed in Sect. 8.1.5. Functions specific to `Integer` include, for example, the calculation operations collected in the protocol `factorization` and `divisibility`.

1. Factorial (`factorial`)
2. Greatest common divisor (`gcd:`)
3. Least common multiplier (`lcm:`)

The methods of the protocol `bit manipulation` make use of the fact that whole numbers are realised within the machine as sequences of bits, and they supply a series of bit-manipulation operations that are especially important for system programming.

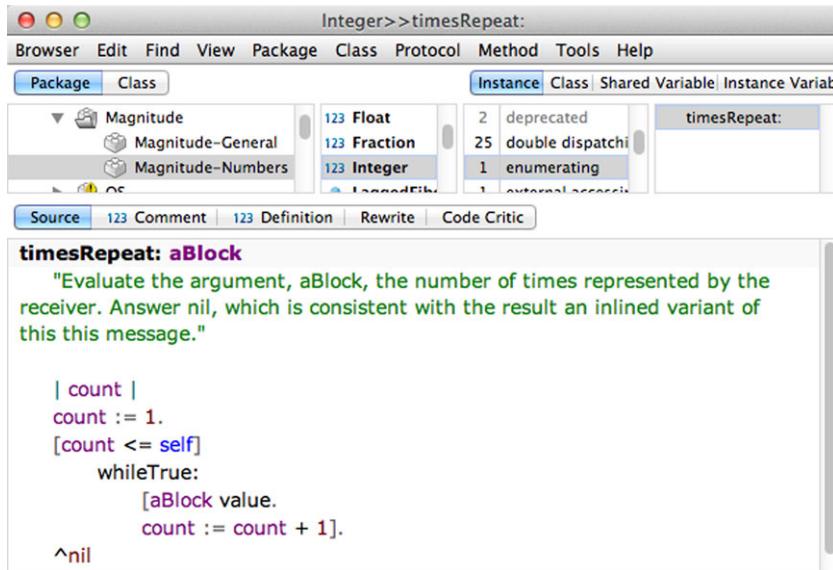


Fig. 8.11 Implementation of timesRepeat :

Conversion Operations

The protocol converting assembles methods that permit the conversion of an Integer number into a different Magnitude object:

1. asFloat and asDouble convert an Integer number into a Real number.
2. asCharacter converts a SmallInteger object into a Character object (see Sect. 8.1.8).
3. The methods asInteger and asRational return the receiver unchanged. Integer numbers are also perceived as rational numbers, although there is no corresponding superclass for Integer and Fraction.

The Method timesRepeat :

Back in Sect. 4.2.1, we got to know so-called count loops as a type of repetitive structure. The simplest form is programmed using the message timesRepeat:, which has an Integer object as a receiver. Figure 8.11 shows the implementation of the method within the class Integer.

This shows that a local variable called count is used as the counter, and that the repeated execution of the argument block (aBlock) is accomplished using the more generalised repetition message whileTrue: (compare Sect. 8.1.7). The message aBlock value evaluates the block. The counter variable count is incremented until it reaches the value of the receiver, which is of course what sets the number of repetitions.

8.1.4 The Classes `Float`, `Double` and `Fraction`

These classes serve to represent non-whole numbers, and thus display certain common characteristics. First, though, we'll examine a few differences. An example of these is the ways in which literals are written.

Literals

Constants for the class `Fraction` are written simply as a division expression, “numerator divided by denominator.” That is, this is not an independent literal representation. A preferred way of writing the expression is to enclose the division expression between parentheses. You can find examples in Figs. 8.9 and 8.6.

In contrast, literals for floating point numbers can always be recognised because the numeral string contains a decimal point. Here are a few examples of valid literals for the class `Float`:

```
8.0 13.3 0.3 2.5e6 1.27e-30 1.27e-31 -12.987654e12
```

The number following the lowercase *e* specifies the power of ten by which the number to the left of the *e* is to be raised. There must be at least one numeral to the right and to the left of the decimal point. The expression containing the exponent is also referred to as *scientific notation* or *exponential representation*.

Literals of the class `Double` use a lowercase *d* instead of an *e*. For example, if you want to store the constant 3.141592653598 as an instance of `Double`, you must write it in the format

```
3.141592653598d
```

Otherwise there is no difference between `Double` and `Float` literals.

Accuracy

`Fraction` objects are always exact, and calculating with fractions always yields exact values. In cases where fractions and floating point numbers both occur in the expression, the result is always a floating point number, and rounding errors can occur.

Instances of the class `Float` are floating point numbers of single precision. They offer an accuracy of approximately six to seven decimal places. The representable number range is between 10^{-38} and 10^{+38} . With instances of `Double`, one achieves an accuracy of approximately 14 to 15 decimal places and a magnitude range between 10^{-307} and 10^{+307} . They are called floating point numbers of double precision.

Both classes understand the message `pi` (see Fig. 8.12).

Because of their internal representation, floating point numbers guarantee a constant *relative* error across their entire representable magnitude. This satisfies a requirement that especially occurs in engineering and scientific calculations. For commercial applications,

Fig. 8.12 The value of π in single and double precision

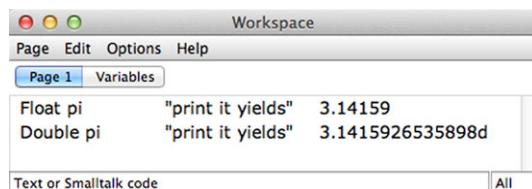
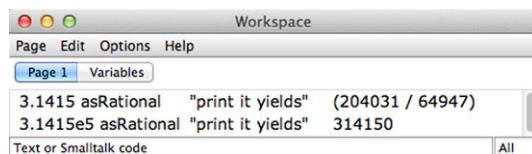


Fig. 8.13 Converting a Float number into a Fraction or Integer object



though, as a general rule, one needs a constant *absolute* error. One might want to always be accurate to two or three decimal places in one's calculations. This is by definition impossible for floating point numbers. One option for achieving absolute accuracy is to represent all currency amounts as whole-number multiples of the smallest unit (for example, one-tenth of a cent), and then to calculate with integers. VisualWorks also uses another number class called `FixedPoint` (see Fig. 6.2), which was already used in Sect. 7.2. Numbers of this class are always accurate to a definable number of places after the decimal point, and are thus the most suitable for commercial calculations. This class, however, is not available in all Smalltalk systems.

Methods

Most methods are inherited from the class `Number` and are discussed in Sect. 8.1.5.

There are two special methods:

`integerPart`, which produces the whole-number portion of a Real number
`fractionPart`, which produces the fractional portion of a Real number

from the protocol `truncation` and `round off` in the class `LimitedPrecisionReal`. You can also use the message `asRational` to convert a `LimitedPrecisionReal` number into a fraction or a whole number (see Fig. 8.13).

8.1.5 Methods Common to All Number Classes

The primary purpose of the class `Number` is to collect the common elements for all types of numbers. This occurs through methods that can be used with all types of numbers and which can be inherited by all subclasses of `Number`.

Mathematical Functions

Figure 8.14 shows the protocol for the instance methods for the class `Number`. This is where mathematical functions, among other things, are found. The figure also offers a



Fig. 8.14 The instance method protocols for the class Number

glimpse of how the tangent function is implemented. First, the message `asLimitedPrecisionReal` converts the receiver object into a `Float` object, and then the method `tan`, which is defined in the class `Float`, is activated for this object. This procedure of first converting the receiver object into a `Float` object is also used for the other trigonometric functions, for log functions and for root and e (`exp`) functions.

The operations `**` and `raisedTo:` both yield the same result: The receiver object is raised to the power expressed in the argument.

The rule for trigonometric functions is that all angles are expressed as radians. That means that Smalltalk assumes that the receiver object for the functions `sin`, `cos` and `tan` is an angle in radians. The same is true for the result of the inverse functions `arcSin`, `arcCos` and `arcTan`. The methods `radiansToDegrees` and `degreesToRadians` (in the protocol `converting`) present a simple option for converting radian measurements to degrees and vice versa. For example, to calculate the cosine of 180° , one can use the expression

```
180 degreesToRadians cos
```

Querying Various Properties

Table 8.2 collects the most important methods used to test numbers for specific properties. They can be found in the `testing` protocol.

Except for `even` and `odd`, all of the methods in Table 8.2 are located in the superclass `ArithmeticValue`.

Table 8.2 Test operations for numbers

Message pattern	Meaning
even	Yields <code>true</code> if the receiver is an even number
odd	Yields <code>true</code> if the receiver is an odd number
positive	Yields <code>true</code> if the receiver is greater than or equal to 0
strictly positive	Yields <code>true</code> if the receiver is greater than 0
negative	Yields <code>true</code> if the receiver is less than 0
sign	Yields 1 if the receiver is greater than 0, -1 if it is less than 0 or 0 if it is equal to 0

Table 8.3 Rounding and truncation

Message pattern	Meaning
Ceiling	Yields the smallest integer that is greater than or equal to the receiver
Floor	Yields the largest integer that is less than or equal to the receiver
Rounded	Yields the integer that is nearest to the receiver
roundedTo: aNumber	Yields the integer multiple of the argument <code>aNumber</code> that is nearest the receiver
Truncated	Yields the integer that is nearest the receiver and that lies on the number line between 0 and the receiver
truncateTo: aNumber	Yields the integer that in the form of an integer multiple of <code>aNumber</code> is nearest the receiver and that lies on the number line between 0 and the receiver

Rounding

The protocol `truncation` and `round off` contains various procedures (rounding and truncation) that can be used to convert non-whole numbers into whole numbers by removing the places after the decimal point. The detailed working of the operations is described in Table 8.3.

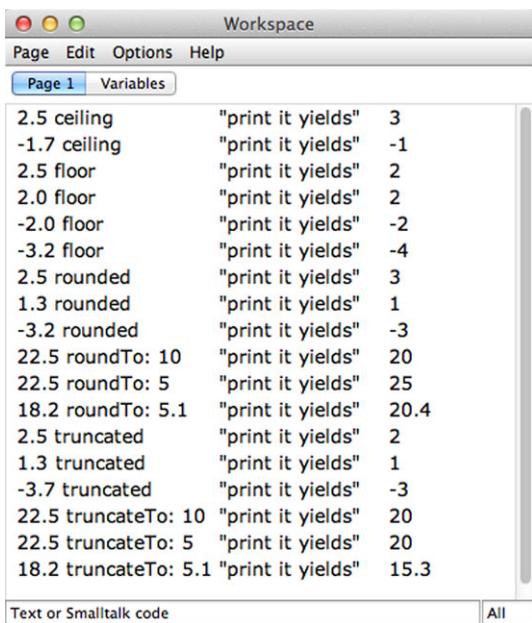
Figure 8.15 shows a few examples of applying these functions.

Intervals

In Sect. 4.2.2, when we discussed using interval run-throughs to create repetitions, we referred to options for defining intervals. The methods to do so are implemented in the protocol `intervals` in the class `Number`. Figure 8.16 shows the simplest method (`to:`) for creating an interval that starts with the receiver and is incremented by 1 until the argument (`stop`) is reached. In the method, the expression

```
Interval from: self to: stop by: 1
```

Fig. 8.15 Examples of the use of methods in the protocol truncation and round off



```

Workspace
Page Edit Options Help
Page 1 Variables
2.5 ceiling      "print it yields" 3
-1.7 ceiling     "print it yields" -1
2.5 floor        "print it yields" 2
2.0 floor        "print it yields" 2
-2.0 floor       "print it yields" -2
-3.2 floor       "print it yields" -4
2.5 rounded      "print it yields" 3
1.3 rounded      "print it yields" 1
-3.2 rounded     "print it yields" -3
22.5 roundTo: 10 "print it yields" 20
22.5 roundTo: 5  "print it yields" 25
18.2 roundTo: 5.1 "print it yields" 20.4
2.5 truncated    "print it yields" 2
1.3 truncated    "print it yields" 1
-3.7 truncated   "print it yields" -3
22.5 truncateTo: 10 "print it yields" 20
22.5 truncateTo: 5  "print it yields" 20
18.2 truncateTo: 5.1 "print it yields" 15.3

```

Text or Smalltalk code All

sends a `from:to:by:` message to the class `Interval`. As Fig. 8.17 shows, this is a Collection class (see Chap. 10). The figure also shows how the class method `from:to:` is implemented. At this point, though, we will not pursue further details of the class `Interval`.

You can use the message `to:by:` to create an interval with any increment you wish. When you use **Inspect it** to evaluate the expression

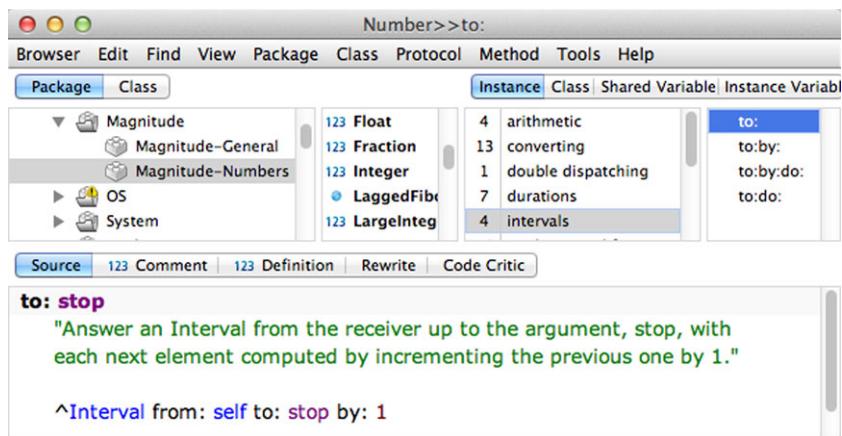


Fig. 8.16 The protocol intervals from the class Number

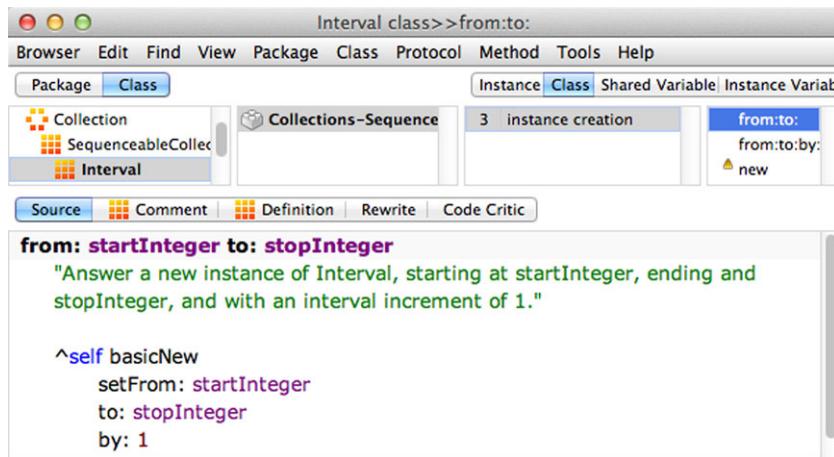
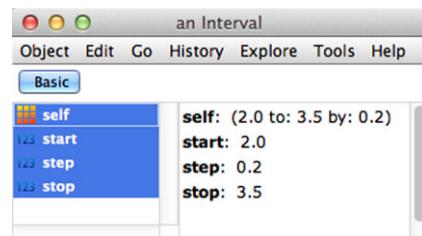


Fig. 8.17 The class `Interval` with the class method `from:to:`

Fig. 8.18 An interval with increments of 0.2



```
2.0 to: 3.5 by: 0.2
```

in the workspace, VisualWorks displays the Inspector windows shown in Fig. 8.18. The last number within the interval is 3.4.

The two messages `to:do:` and `to:by:do:` both create an interval in the same way, as do messages without the keyword `do:.` In addition, the block provided as an argument after the `do:` keyword is evaluated for every element of the interval. You can find an example in Sect. 4.2.2.

8.1.6 Mixed Expressions

In the previous sections, we've occasionally discussed the problem of determining to which number class the result of an arithmetic expression belongs when its operand objects belong to different classes. At this point, we'll describe the rules that form the basis for the evaluation of such expressions in Smalltalk.

The first rule is:

If two objects taking part in a numeric operation belong to different classes, one of the objects must be converted into the other's class.

In Smalltalk, this conversion process is referred to as *coercion*. The examples we've shown have already discussed the fact that, in the expression `2 * 3.5`, the `SmallInteger` number 2 is converted to a `Float` object before the two numbers are multiplied; the result is a `Float` number.

The second rule determines into which class objects in mixed expressions are to be converted:

Determine which object belongs to a class with the highest numeric generality. Convert all other objects to that class.

The generality concept in Smalltalk is designed so that each class must be able to convert their instances into an instance of the class with the next higher generality, and to do so in such a way that the instance's numeric value is maintained as nearly as possible. The number classes described in this chapter are ordered according to generality in the following way (starting with the highest):

```
Double  
Float  
Fraction  
LargePositiveInteger LargeNegativeInteger  
SmallInteger
```

You can always perform a “lossless” conversion of a `SmallInteger` number into a `LargePositiveInteger` or a `LargeNegativeInteger`, depending on the leading sign. When you convert `Integer` or `Fraction` numbers to `Float` or `Double`, though, you may lose accuracy if the entire number should contain more decimal places than are available in the `Real` class you are converting to.

It may also happen that a `LargePositiveInteger` number, should it be large enough, cannot be converted into a `Real` number at all. For example, the arithmetic expression

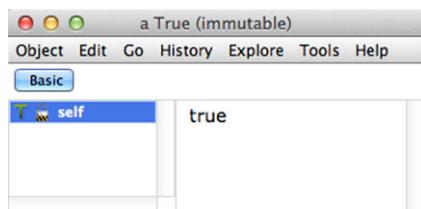
```
170 factorial + 1.0d
```

can be evaluated, because 170 can still just be represented as a `Double` number. The result of the evaluation is

```
7.257415615308d306
```

An attempt to evaluate the expression

Fig. 8.19 true is an instance of the class True



```
171 factorial + 1.0d
```

ends with the error message that 171 can no longer be expressed as a `Double` number.

Whenever `Real` numbers appear in mixed expressions, you must always be aware of the problems that arise from their limited accuracy and their limited range of magnitude. When you require precision, you should not use those numbers. The decision by Smalltalk's developer to give the `Real` classes a higher generality than the `Fraction` class appears arbitrary against this background.

8.1.7 Truth Values

Back in Sect. 2.3.3, as part of the discussion surrounding the general solution of a quadratic equation, we encountered the problem of having to make case-by-case distinctions in an algorithm. In Smalltalk, these are sometimes written as conditions in the form of comparison expressions (for example, `a > 0`), to which a message like `ifTrue::`, `ifFalse::`, etc. is sent. The evaluation of a comparison expression always yields a Boolean object. There are precisely two of them, which are characterised by the pseudo-variables `true` and `false`. Of these pseudo-variables, `true` is the only instance of the class `True`, and `false` is the only example of the class `False` (see Fig. 8.1). If one examines the expression `5 = 5`, for example, one sees in the window title of the Inspector (see Fig. 8.19) that it is an instance of the class `True`.

The class `Boolean` serves both classes as a common superclass, which implements the logical operations equivalence (`equiv::`) and antivalence (`xor::`), while the Boolean operations conjunction (`&`), disjunction (`l`) and negation (`not`) are implemented in `True` and `False` respectively.

As a placeholder for the others, implementation of the conjunction will be shown in Figs. 8.20 and 8.21. In the expression `a & b`, when the receiver `a` has the truth value `true`, the `&` method of the class `True` is executed. The result of the conjunction then depends solely on the argument `b`, for which reason the method then returns the argument as its result. In case the receiver `a` has the truth value `false`, the `&` method of the class `False` is executed. The result of the conjunction is in this case independent of the value of the argument; in all cases it is `false`.

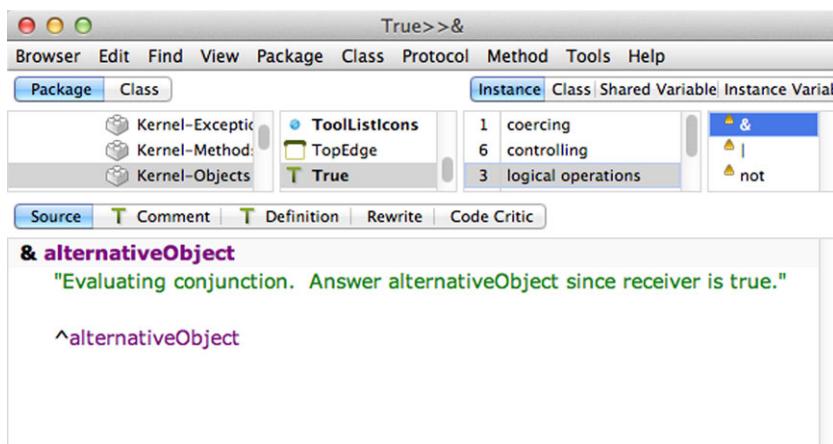


Fig. 8.20 Implementation of the conjunction in the class True

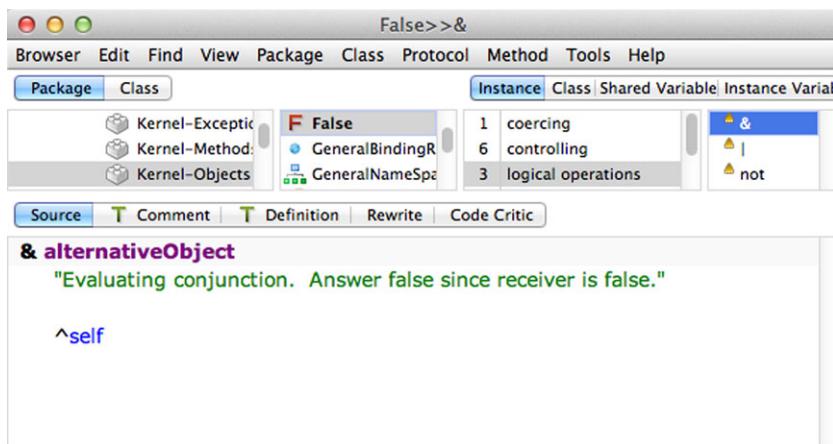


Fig. 8.21 Implementation of the conjunction in the class False

Besides the messages for making case-by-case distinctions, the protocol controlling (see Fig. 8.22) of the two classes, also contains two alternative methods for the conjunction (`and:`) and the disjunction (`or:`). The following example will be used to briefly explain their meaning:

```

| a b |
a := 4.
b := 0.

```

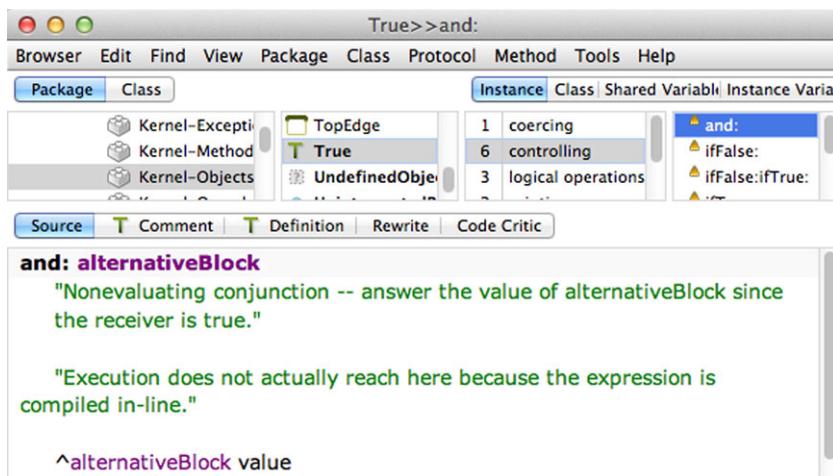


Fig. 8.22 Implementation of the *non-evaluating* conjunction in the class True

```

((b ~= 0) \& (a/b >2))
ifFalse: [Transcript show: 'it does not work this way']

```

When we try to execute the program shown above, the following problem arises during the evaluation of the `ifFalse:` condition: before the message `\&` is sent to the receiver `(b ~= 0)`, its argument `(a/b >2)` is evaluated, which leads to a program abort because `b=0`. On the other hand, the result of the And-link is `false` anyway, because `b` is equal to 0. To determine this result, it's not even necessary to evaluate the `\&` message. For cases of this type, you can use the *non-evaluating* methods `and:` and `or:.` For the argument, they expect a block that is evaluated within the method only when it is necessary. We can rewrite the previous program in this way:

```

|a b|
a := 4.
b := 0.
((b ~= 0) and: [(a/b >2)])
ifFalse: [Transcript show: 'it works this way']

```

Since the receiver of the message `and:` is `false` in this case, the block transmitted as the argument `[(a/b >2)]` is not evaluated, and therefore no division by 0 is executed.

The implementation of the methods for case-by-case distinctions is explained here using as an example the method `ifTrue:ifFalse:` from the class `False`, as shown in Fig. 8.23. Because the receiver of the message is `false` when the method in `False`

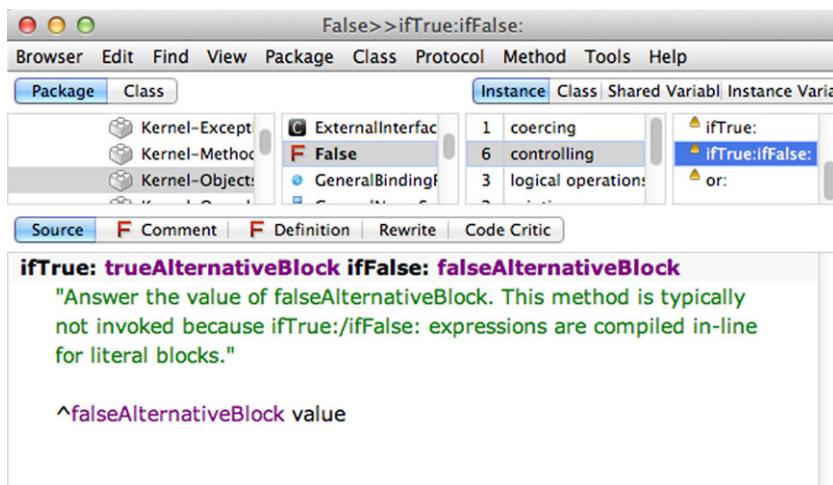


Fig. 8.23 Implementation of the method `ifTrue:ifFalse:` in the class `False`

is activated, the argument block that is transmitted after the keyword `ifFalse:` must be evaluated. (The placeholder for the argument block is indicated in the method as `falseAlternativeBlock`.) For that reason, the message `value` is sent to the block, which affects the evaluation.

The other three messages also function according to the pattern described here.

8.1.8 Characters and Character Strings

The Class `Character`

The instances of the class `Character` represent typeable and non-typeable characters. The typeable characters are especially those that can be entered via the keyboard: letters, numerals and a variety of special characters such as parentheses and brackets, punctuation marks, etc.

In Sect. 3.2, we noted that the representation of typeable characters as literals consists of a dollar sign followed by the typeable character.

For various non-typeable characters, you can send messages to the class `Character`, which produces the characters. You can determine which characters these are from the class protocol accessing untypeable characters (see Fig. 8.24). For example, the message

Character space

can create the `Character` object for an empty space.

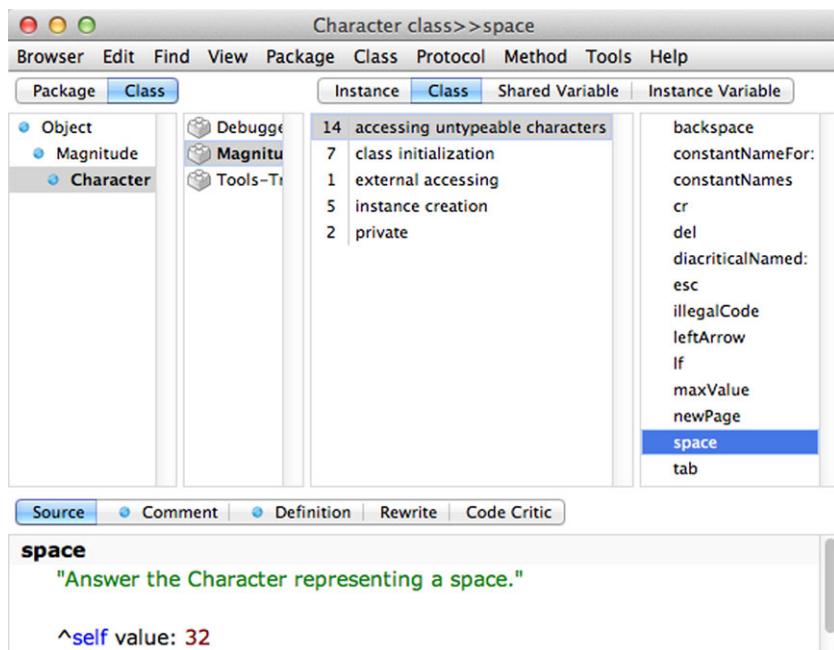


Fig. 8.24 Class methods for creating untypeable characters

Each character is coded internally using a whole number. Figure 8.24 shows, for example, that the number 32 is used for the space.

Character codes in the version of VisualWorks that served as the basis for this book are whole numbers between 0 and 65535 (16rFFFF). The assignment of codes to characters occurs according to the internationally standardised Unicode Character Code Standard.

You can use the message `asInteger` from the method protocol `converting` to determine which code belongs to a `Character` object. Thus, the evaluation of the expression

```
Character space asInteger
```

yields the value 32. Conversely, as shown in Fig. 8.24, the class method `value:` can be used to produce a character using a particular code.

The protocol `converting` provides two additional methods that allow you to convert a lowercase letter into uppercase (`asUppercase`) and vice versa (`asLowercase`).

The method `digitValue` converts a character into a numeric equivalent. This is especially interesting for the characters \$0 through \$9. For example, the expression

```
$5 digitValue
```

produces the `SmallInteger` object 5.

Another important characteristic of objects in the class `Character` is that, as a result of the internal coding using whole numbers, the comparison operations (`<`, `=`, etc.) can also be used on them. When the character codes were set up, the ascending numbers paralleled the order of the alphabet, that means, for example, that `$f < $g`. Uppercase letters are “smaller” than lowercase, so `$A < $a`. And the numeric order applies to the decimal numerals `$0` through `$9`.

The protocol testing contains some useful test methods that allow you to ask, for example, whether a character is

1. a vowel (`isVowel`)
2. a letter (`isLetter`)
3. a numeral (`isDigit`)

The Class `String`

Section 3.2 also introduced character-string literals, which are instances of the class `String`. Figure 8.1 shows that this is a subclass of `Collection`. Chapter 10 deals with common characteristics of the collection classes. At this point, we’ll anticipate descriptions of a few special methods that are interesting for dealing with character strings.

A character string represents a container for characters. In other words, the characters are components of a character string. As with arrays, each component can be accessed using an index. The expression

```
'hallo' at:2
```

produces the character `$a` as a result. Using the message `at :put :`, one character within a character string can be replaced with another:

```
'hallo' at:2 put: $e
```

If you use **Print it** to execute the program

```
| s |
s := 'hallo'.
```

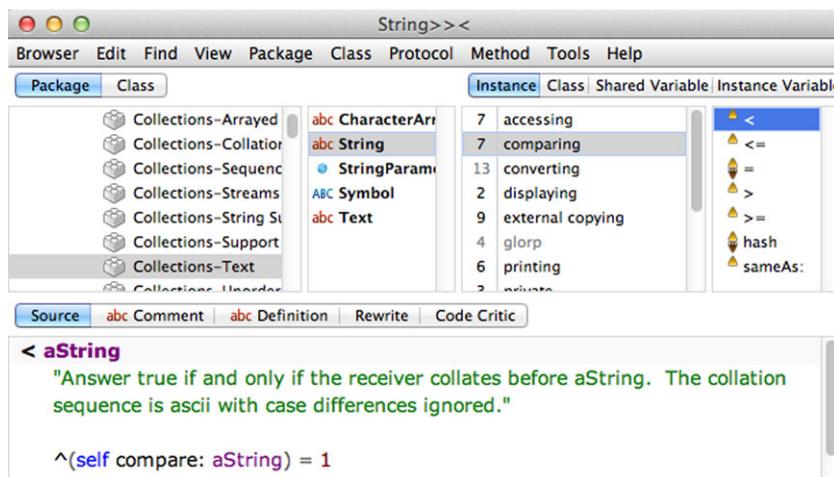


Fig. 8.25 Comparison methods for the class `String`

```
s at: 2 put: $e.
```

in the workspace, you receive the character string '`hello`'.

► **Note:** Beginning with Version 7 of VisualWorks, literals by nature cannot be changed (so-called *immutables*). That means that the Smalltalk sequence we just described will lead to an exception. Using the message `copy`, however, one can create a copy of a `String` literal, which then represents a “normal”, mutable object. The expression

```
'hallo' copy at:2 put: $e
```

yields the desired result.

Figure 8.25 shows that the class `String` has its own comparison methods. None of the methods except “`=`” are case sensitive. Thus, the comparison

```
'karla' < 'Karlo'
```

for example, produces `true`, even though the character `$k` occurs in the code series after the character `$K`.

Table 8.4 Examples of pattern matching

Message pattern	Evaluation yields...
'xyz' match: 'Xyz'	true
'x#z' match: 'xyz'	true
'x*z' match: 'x abc? z'	true
'*x' match: 'x'	true
'#x' match: 'x'	false

In contrast, the method “`=`” performs an exact comparison of the two character strings. You can test for sameness without case-sensitivity using the method `sameAs::`. The expression

```
'String' = 'string'
```

yields `false`, while

```
'String' sameAs: 'string'
```

yields `true`.

Pattern Comparison

Beyond merely comparing character strings, you can also perform *pattern matching*. Two special wildcards are available for this function:

`$#` represents exactly one instance of any character

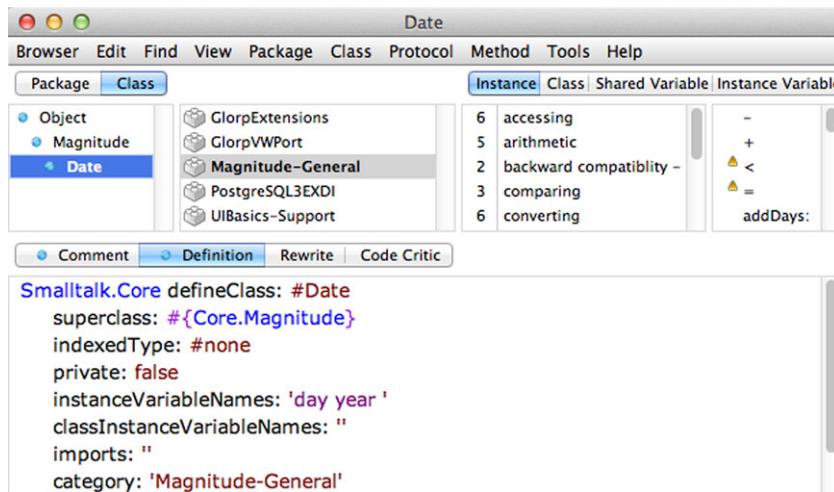
`$*` represents any series of characters (including a blank space)

One pattern-match method is called `match`: Table 8.4 shows a few applications. The first line shows that the match is not case sensitive. The method tests whether the character string sent as an argument corresponds to the pattern represented by the receiver. The wildcards are effective in the receiver character string, not in the argument string. If the pattern match is to be case sensitive, you can use the method `match:ignoreCase::`. A Boolean object is expected as an argument after the second keyword. The evaluation of the expression

```
'xyz' match: 'Xyz' ignoreCase: false
```

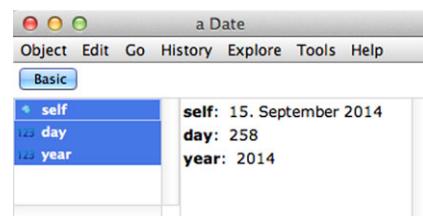
yields `false`.

You can use the messages `asLowercase::` and `asUpperCase::` to convert character strings to lowercase or uppercase.

**Fig. 8.26** The class Date**Fig. 8.27** September 15, 2014

as an instance of the class

Date



8.1.9 Date and Time

Two classes—Date and Time—are available in Smalltalk for representing date and time; as Fig. 8.1 shows, they are subclasses of the class Magnitude.

The Class Date

A date is represented by a year and by the number of the date counted from the first day of the year. The instance variables `year` and `day` serve this purpose (see Fig. 8.26). Figure 8.27 shows an example the representation of September 15, 2014.

There are several options for using the class methods of the class `Date` to create a date. The current date is supplied by the expression

```
Date today
```

One frequently used option is to use the message `readFromString:` to convert a character string into a Date object. The character string can contain the date in various formats, for example:

Table 8.5 Important class variables in the class Date

Class variable	Structure	Meaning
DaysInMonth	Array of integers	The number of days in a month
FirstDayOfMonth	Array of integers	The number of the first day of a month calculated from the first day of the year
MonthNames	Array of symbols	The names of the twelve months
SecondsInDay	Integer	Number of seconds in a day
WeekDayNames	Array of symbols	The names of the days of the week

```
Date readFromString: 'December 1, 2002'.
Date readFromString: '1 December 2002'.
Date readFromString: '12.1.2002'.
Date readFromString: '12-1-2002'.
```

At this point, we will not discuss the options for setting region-specific date formats.

You can also specify a date by entering a date, a month and a year:

```
Date newDay: 1 monthNumber: 12 year: 2002.
Date newDay: 1 month: #December year: 2002.
```

In the second case, the name of the month is supplied as a symbol.

There are a number of class variable available in the class Date where calendar information is stored. Table 8.5 shows their structure and meaning. Figure 8.28 shows the class method initialize, which sets the class variables. If necessary, it would of course be possible to modify this method.

Date objects respond to the binary comparison methods (=, ~=, <, <=, >, >=); the message “<” is interpreted as “earlier than” and “>” as later than.

You can also use date information to calculate:

- The message addDays : adds a number of days to a date.
- The message subtractDays : subtracts a number of days from a date.
- The message subtractDate : calculates the difference (expressed as a number of days) between the receiver and the argument.

Furthermore, a Date object also knows (among other things):

- How its day number, month number and year are to be calculated (protocol accessing).

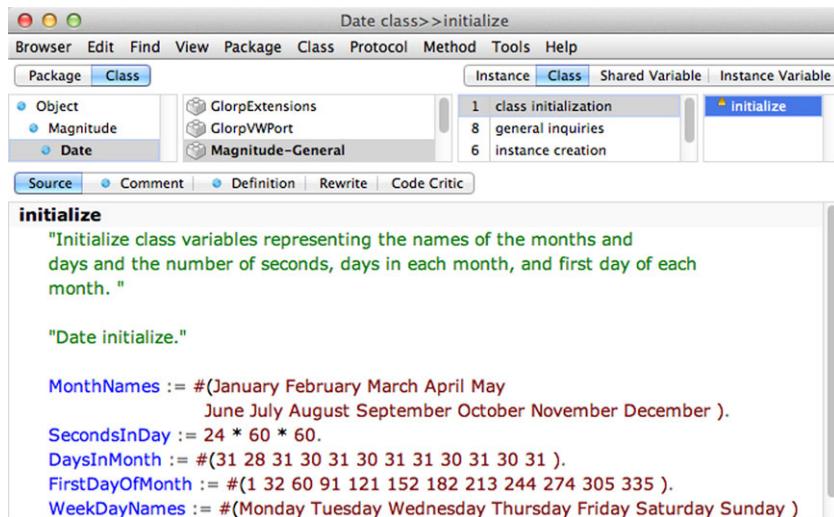


Fig. 8.28 Initialising the class variables for the class Date

- The names of the day of the week and the month (protocol accessing).
- Whether it occurs in a leap year (protocol accessing).
- How many days or seconds lie between itself and January 1, 1901 (protocol converting).
- How many days are in its month or its year, and how many days remain in its year (protocol inquiries).

The Class Time

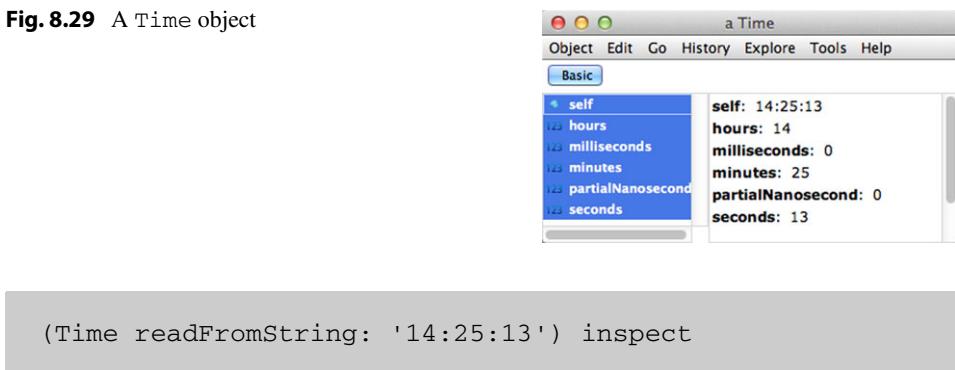
One instance of the class Time represents the number of seconds that have passed since midnight, where the time is shown as whole numbers in the instance variables hours, minutes and seconds. You can use the methods bearing the same names as the variables located in the protocol accessing to access the instance variables.

As was true for the class Date, you can create instances of Time using class methods from the protocol instance creation.

- now produces the current time.
- fromSeconds: expects a whole-number argument, which is interpreted as the number of seconds that have passed since midnight.
- readFromString: expects a character string, which must be supplied in the format

```
'Hour:Minute:Second:'
```

For example, the evaluation of the expression

Fig. 8.29 A Time object

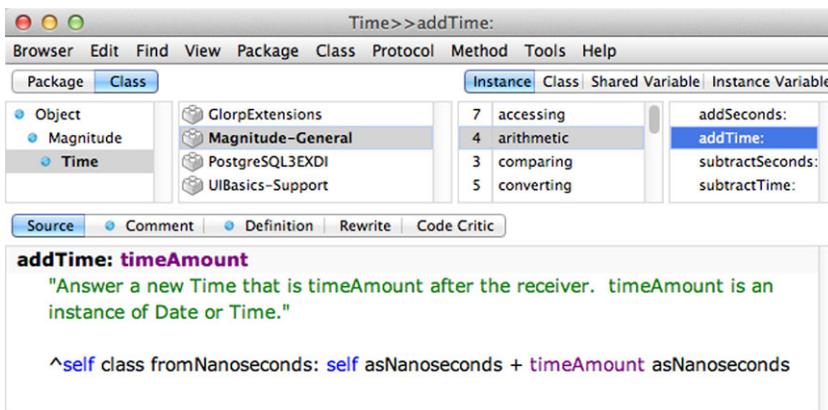
yields the Time object shown in Fig. 8.29.

Time expressions can be compared in the same way as date expressions.

The messages `asSeconds`, `asMilliseconds` and `asNanoseconds` (protocol converting) permit the conversion of a Time object into a whole number that shows the time units that have passed since midnight. The message `asNanoseconds`, for example, is used for calculating times. The instance methods `addTime:` (see Fig. 8.30) and `subtractTime:` from the protocol `arithmetic` can be used for these calculations.

8.2 Abstract and Concrete Classes

When you look at the class hierarchy shown in Fig. 8.1 in the light of the matters discussed in the previous sections of the chapter, you might notice that not all classes seem to be used to create instances of themselves. For example, each whole number is an instance of exactly one of the classes `SmallInteger`, `LargePositiveInteger` or

**Fig. 8.30** Method for adding times

`LargeNegativeInteger`. Their common superclass `Integer`, though, has no class “of its own.” A class from which no instances can be created is called an *abstract* class. Classes from which instances can be created are called *concrete*.

The purpose of an abstract superclass is to gather together common aspects of its subclasses and to make them available to them through inheritance. (See Sect. 6.1 on this topic too.) Both the structure (data, represented by instance variables) and the behaviour determined by methods can be inherited. But don’t assume that the reverse is true, that classes that possess subclasses must necessarily be abstract. For example, the class `String` has a subclass `Symbol`, which of course inherits methods from `String`. Yet both classes are concrete classes.

The class `Number` is a further example of an abstract class. Among other things, the class has available in the protocol `mathematical functions` a series of mathematical functions for all types of numbers.

The abstract class `ArithmeticValue` is the abstract superclass of all classes with objects that understand the four basic arithmetic calculation methods. These include not just instances of the subclasses of `Number`, but also those of the class `Point`, which is also a subclass of `ArithmeticValue`. In order for arithmetic operations to be able to be carried out with instances of the various subclasses of `ArithmeticValue`, they must be able to put into practice the coercion concept that was described in Sect. 8.1.6.

The Class `Magnitude`

In order to understand abstract classes better, let’s look more closely at the class `Magnitude`. Figure 8.31 shows a section of the class comment for the class `Magnitude` that indicates that this class is the common, abstract superclass of all classes for the objects of which a linear order has been defined on the basis of the comparison operations “`<`” and “`=`”. Figure 8.1 shows that, besides `ArithmeticValue`, the classes `Date`, `Time` and `Character` are also subclasses of `Magnitude`. Both comparison operations are implemented in those classes. If one should want to create one’s own classes and add them to the class hierarchy as subclasses of `Magnitude`, it would be necessary to realise the comparison operations in them as well.

Figure 8.32 shows that the class `Magnitude` itself contains a `<` method. The evaluation of the expression

```
^self subclassResponsibility
```

in the method body would, however, lead to a program abort. The debugger would appear with the error message

Unhandled Exception: My subclass should have overridden one of my messages.

This says nothing more than that the `<` method from the class `Magnitude` was executed only because the `<` method was not implemented in one of the subclasses. Since `Magnitude` as an abstract class has no instances of its own, the `<` method from

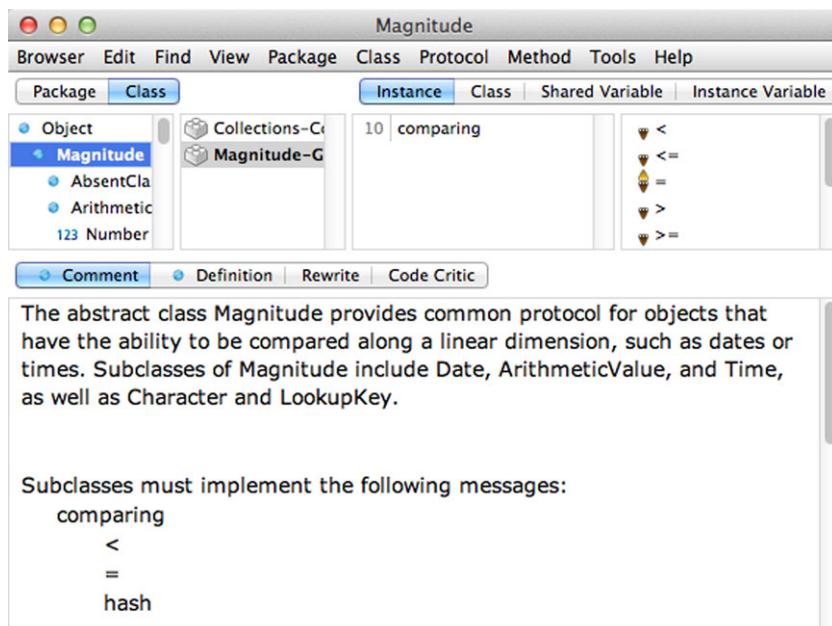


Fig. 8.31 Section of the class commentary of the class Magnitude

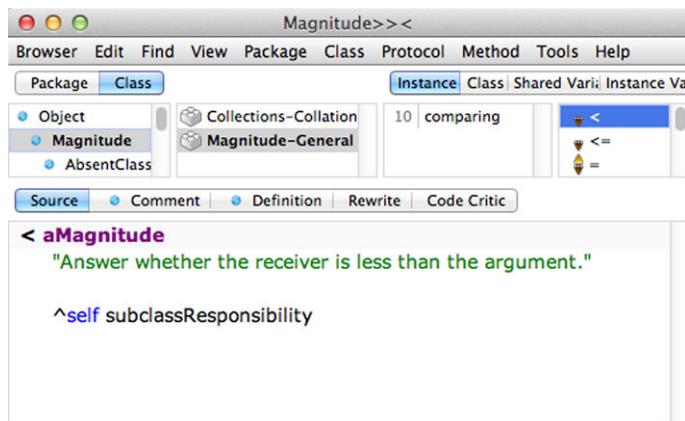


Fig. 8.32 The abstract method <

Magnitude can be activated only by sending a < message to an instance of a subclass that has no < method of its own. On the inheritance principle, the < method of the superclass will be executed.

That is because the < method in the class Magnitude is defined as a so called abstract method. An abstract method in an abstract class serves as a placeholder to indicate that this method must be implemented—or we might say “concretised”—in all

subclasses. Implementing the `<` method as an abstract method in the class `Magnitude` means that the method is essential for all `Magnitude` objects, but that it can only be implemented in the concrete subclasses. From the point of view of the class `Magnitude`, the situation could also be explained in an object-oriented way:

All classes that inherit from me, the class `Magnitude`, must define the method “`<`”. I myself have no idea how to do this. But there’s an error if the method is not defined. And it’s my role to point that out.

Unlike in Java, for example, Smalltalk has no syntactic option for marking a method as abstract. Instead, the convention applies that abstract methods are always implemented like the `<` method in `Magnitude`. The method “`=`” is also defined as an abstract method.

8.3 Generic Methods

Now let’s examine the implementation of the remaining comparison methods in the class `Magnitude`. As an example, let’s look at the `<=` method:

```
<= aMagnitude
  "Answer whether the receiver is less than or equal to
   the argument."
  ^self > aMagnitude) not
```

This method is not abstract, but is fully programmed using the method “`>`”, the result of which is negated. The `>` method itself looks like this:

```
> aMagnitude
  "Answer whether the receiver is greater than the
   argument."
  aMagnitude < self
```

It is defined using the abstract `<` method. Methods belonging to an abstract class that are defined directly (`>`) or indirectly (`<=`) using an abstract method are also referred to as *generic* methods.

Notice that all of the other comparison operations have been defined as generic methods in the class `Magnitude` and are in general valid for all `Magnitude` objects. Additional generic methods are `between:and:, min:, and max::`.

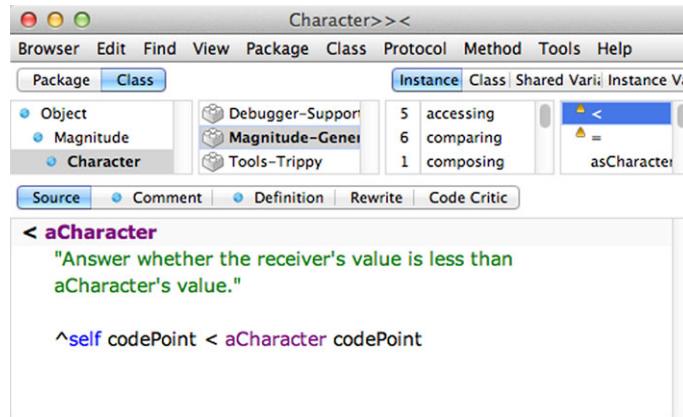


Fig. 8.33 The protocol comparing in the class Character

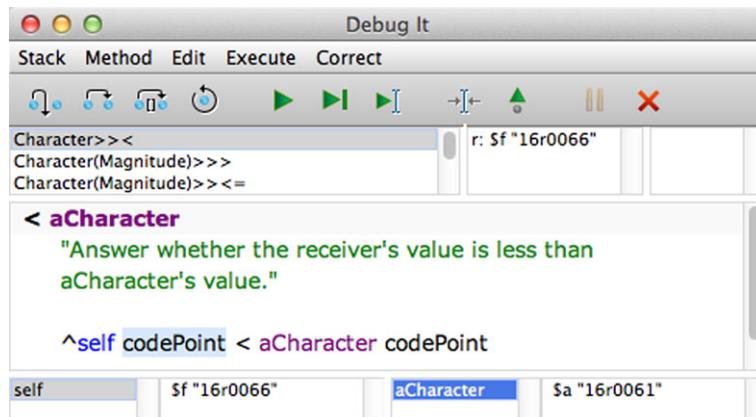


Fig. 8.34 Snapshot of the execution of the expression "\$a <= \$f"

An important use of abstract classes lies in the fact that generic methods can be implemented once in the abstract class on the basis of abstract methods that will be concretised in the subclasses; for that reason, the methods do not need to be implemented in the subclasses.

In order to illustrate method activation using a generic method, we'll examine the test of two Character objects with respect to “ \leq ”. Figure 8.33 shows us that the class Character implements the comparison operators “ $<$ ” and “ $=$ ” but no others, just as we would expect from a subclass of Magnitude.

When we use the debugger to evaluate the expression $\$a \leq \f , repeated use of the **Step** and **Step into** keys produces the method stack shown in Fig. 8.34.

In this case, the fourth line represents the activation in the workspace of the nameless method that sends the message `<= $f` to the `Character` object `$a`. Since the class `Character` does not have its own `<=` method, it uses the method inherited from the superclass `Magnitude`. The third line of the method stack shows this. The representation

```
Character(Magnitude) >><=
```

is to be interpreted in the following way: For an object of the class `Character`, the method “`<=`” inherited from its superclass `Magnitude` is executed. This generic method now activates in turn (see above) the generic method “`>`”. The second line of the method stack shows this. In the `>` method, finally the `<` method is activated that was concretely programmed in the class `Character`; this is shown in the first line.

As an aside, note that the implementation of the `<` method, shown in the middle section of Fig. 8.34, shows that the comparison of `Character` objects by sending the message `codePoint` is based on the comparison of whole numbers. (See the description of the class `Character` in Sect. 8.1.8.)

8.4 Polymorphism

In object-oriented terminology, one says that generic methods are polymorphic. That term means that the activation of one and the same generic method in an abstract class leads to the execution of different concrete methods depending on which concrete subclass the receiver of the message that leads to the activation of the generic method of the abstract superclass belongs to.

To make this clearer, let's look at the class `Date`, which, like `Character`, is a subclass of `Magnitude`. If we again use the generic method “`<=`” to evaluate the expression

```
Date today <= Date today
```

the debugger displays the method stack shown in Fig. 8.35. On a purely superficial level, this figure differs from that shown in Fig. 8.34 in that everywhere that the earlier figure showed `Character`, the later one shows `Date`, because the receiver of the message `<= Date today` is an instance of the class `Date`. Above all, this means that the first line of the activation of the concrete `<` method shows the class `Date`. The generic methods inherited from `Magnitude`, the activation of which is shown in the second and third lines, though, are exactly the same as those shown in Fig. 8.34 for the comparison of two `Character` objects. That is, the activation of one and the same generic method “`<=`” from `Magnitude` leads in the one case to the execution of the `<` method from the class `Character` and in the other to the activation of the identically named method from the class `Date`.

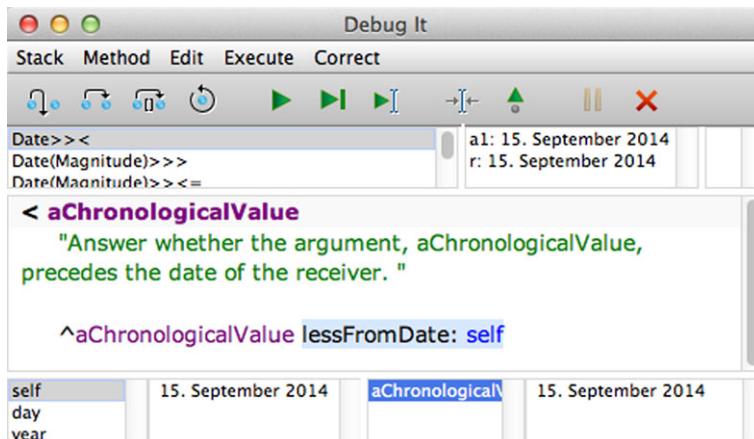


Fig. 8.35 Snapshot of the execution of the expression "Date today <= Date today"

This is one form of Smalltalk's so-called polymorphism. A further, more general form of polymorphism lies simply in the fact that objects belonging to different classes understand the same messages, although they activate different methods. Thus, we have just seen the example where instances of the two classes `Date` and `Character` both understand the message "`<`", yet each executes its own method when it receives the message.

It's part of good programming style to ensure that methods from different classes that bear the same name perform at least somewhat similar actions.

Polymorphism—along with the class principle, inheritance and information hiding—counts as one of the distinguishing characteristics of object-oriented programming languages.

8.5 Case Study: Quadratic Equations

In Sect. 2.3, we wrote a process in the workspace in the form of a Smalltalk program for solving quadratic equations. The program queried the user via a dialogue window for the coefficients that defined the equation, and provided results in the same manner. At that point of the discussion, we were concentrating on examining algorithms, specifically on how to develop an algorithm characterised by multiple case-by-case distinctions, and how to turn that algorithm into a program.

Now we will examine the problem of solving quadratic equations from a more object-oriented perspective. We'll assume that a class exists that we'll call `QuadrEquat`, the instances of which are quadratic equations to which we can send the message `solveYourself`. The object then determines its solution and notes it in one of its instance variables. Additional messages should allow doing things like determining whether the equation has a solution and, if so, how many; in addition, one should be able to access the individual solutions. There is no longer any interaction with the user.

When we introduce a new class, the first thing we have to do is to determine what properties characterise its objects. A quadratic equation in the form

$$ax^2 + bx + c = 0$$

is determined on the one hand by its coefficients a , b and c , and on the other by its solution. That means that the class `QuadrEquat` requires four instance variables.

But now we're faced with the question of, in what format should the solution be stored in the instance variable that's intended for it. A quadratic equation can have no solution or one, two or an infinite number of real solutions.¹ In other words, the solution to a quadratic equation is itself a complex structure. That suggests that we might want to consider solution objects as members of their own class, which we still have to create. We'll call this class `Solution`.

In the class `QuadrEquat`, the instance variables for the coefficients should be called `a`, `b` and `c`. The instance variable `solution` will serve the purpose of accepting the solution, that is, one of the objects of the class `Solution`.

On the condition that the classes `QuadrEquat` and `Solution` have been created, including the methods required for determining the solution, the following test could be executed in the workspace.

```
| eq |
eq := QuadrEquat a: 2.0 b: 3.0 c: -5.0.
eq numberSolutions > 0
ifTrue:
[eq numberSolutions = 1
ifTrue:
[Transcript cr;
show: 'Equation has one solution: ';
eq solutionOne printString]
ifFalse:
[eq numberSolutions = 2
ifTrue:
[Transcript cr;
show: 'Equation has two solutions: ';
eq solutionOne printString,
' and ',
eq solutionTwo printString]
ifFalse:
[Transcript cr; show:
```

¹We will continue to ignore complex solutions.

```

        'Equation has infinitely many solutions'
    ]]]
ifFalse:
    [Transcript cr;
     show: 'Equation has no real solution']

```

When we do this, we assume the existence of the following methods in the class `QuadrEquat`:

- Class methods `a:b:c:`
 - creates an instance of the class `QuadrEquat` with the specified coefficients, and
 - causes the calculation of the solution.
- The message `numberSolutions` (instance method) yields the number of solutions.
- The messages `solutionOne` and `solutionTwo` (instance methods) yield the two possible solutions.

The next section explains how these methods are implemented.

8.5.1 The Class `QuadrEquat`

Figure 8.36 shows that for this case study, we created a package with the name `QuadEquats` and a namespace called `QuadEqNs`. One can see that

- The class `QuadrEquat` has been created in the namespace `QuadEqNs`.
- The class `Object` is defined as the superclass of `QuadrEquat`.
- The instance variables are called `a`, `b`, `c` and `solution`.

Methods of the Class `QuadrEquat`

The methods will not be shown in the System Browser, but will merely be presented as text. There is, however, in Smalltalk no designated syntax for this purpose. If one were to remove a method definition from its representation in the Browser, one would lose its context, that is, it would no longer be clear to what class the method belongs, or whether it is an instance or a class method. For that reason, we'll use a frequently used notational system:

- “`QuadrEquat>solveYourself`” indicates that the method `solveYourself` is an instance method of the class `QuadrEquat`.
- “`QuadrEquat class>a: aNumber b: aNumber2 c: aNumber3`” defines the method `a:b:c:` as a class method of the class `QuadrEquat`.

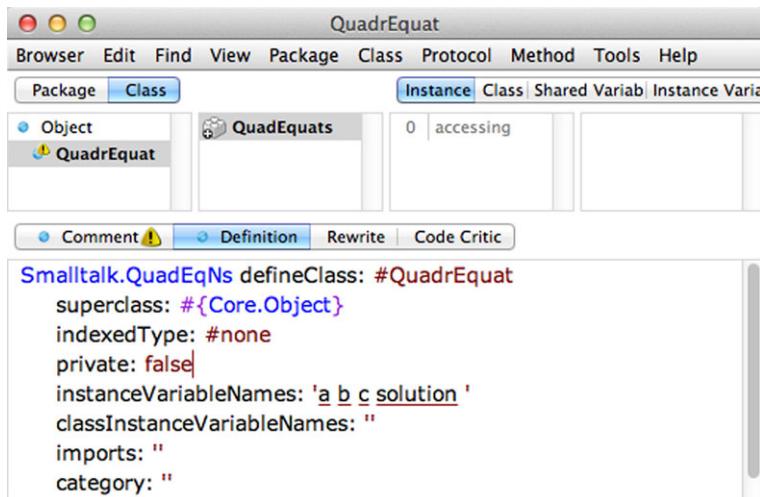


Fig. 8.36 The class QuadrEquat

Based on the above, we write the second method as follows:

```

QuadrEquat class>>a: aNumber b: aNumber2 c: aNumber3
    "creates an instance of the receiver with the
     coefficients aNumber, aNumber2, and aNumber3"

    ^self new a: aNumber b: aNumber2 c: aNumber3

```

Notice that this class method makes use of an instance method with the same name: `a:b:c::`. This is not a problem (and is entirely customary in Smalltalk), because the VM can always assign the message correctly, because the receiver is either the class or an instance of the class.

```

QuadrEquat>>a: aNumber1 b: aNumber2 c: aNumber3
    "defines the quadratic equation (receiver) by
     assigning the coefficients and calculating
     the solution"

    a:= aNumber1.
    b:= aNumber2.
    c:= aNumber3.

```

```
self solveYourself

QuadrEquat>>numberSolutions
    "supplies the number of solutions for the quadratic
     equation (receiver)"

^self solution numberSolutions
```

In the above, the method `numberSolutions` sends a message with the same name to the solution object stored in the instance variable `solution` (see below). You can also say that the `QuadrEquat` object *delegates* the task to another object.

```
QuadrEquat>>solveYourself
    "calculates all real solutions to the quadratic
     equation (receiver)"

self solution: (a = 0
               ifTrue: [self solveLinearEquation]
               ifFalse: [self solveQuadraticEquation])
```

We'll set aside the methods

- `solveLinearEquation` and
- `solveQuadraticEquation`

for the moment until it's clear what solution objects are actually supposed to look like.

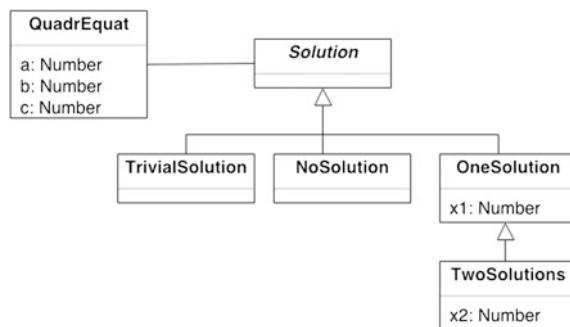
Once again, the two following methods use delegation to solve their “problem”:

```
QuadrEquat>>solutionOne
    "produces the first solution to the quadratic
     equation (receiver)"

^self solution x1

QuadrEquat>>solutionTwo
```

Fig. 8.37 Class system for quadratic equations



```

"produces the second solution to the quadratic
equation (receiver)"

^self solution x2
  
```

And, finally, the get and set methods for the instance variable `solution`:

```

QuadrEquat>>solution
  ^solution

QuadrEquat>>solution: aQuadEqSolution
  solution := aQuadEqSolution
  
```

8.5.2 Classes for Solution Objects

Thinking a little more fully about the structure of the solutions to quadratic equations leads to the realisation that the case where a quadratic equation has two real solutions—which might be called the instance variables `x1` and `x2`—is just one of many possibilities. In case the equation has no solution at all, it makes no sense to create a solution object with two instance variables. Depending on the character of the solution, different types of solution objects may be necessary; thus the instances ought to be from different classes.

Figure 8.37 shows a possible solution for classes of quadratic equations and their solutions. It shows a class diagram that is suggested by the Unified Modeling Language (UML²) and which contains the following information:

- The class `QuadrEquat` contains the instance variables `a`, `b`, `c` and `solution`.

²See also Sect. 14.5.

- After each instance variable, is shown the class of objects that are to be stored in that instance variable. These classes are called *attribute classes*.
 - The attribute class for `solution` is `Solution`. Since that class is part of the diagram, the instance variable `solution` is indicated as an *association* by a line connecting it to the attribute class.
 - The italic font in the name of the class `Solution` indicates that this is an abstract class.
 - The abstract class `Solution` has the subclasses:
 - `TrivialSolution`
 - `NoSolution`
 - `OneSolution`
- And the last class has the subclass `TwoSolutions`.

The four concrete subclasses for `Solution` reflect the fact that quadratic equations can have four different types of solutions. The decision to make the class `TwoSolutions` a subclass of `OneSolution` was based on a desire to reuse structures (the instance variable `x1` is inherited) and behaviour (inheritance of methods, see below). Interpreting the relationship between these two classes as “is a” does not make sense in this instance.

Definition of the “Solution Classes”

At this point, we’re going to show a different procedure for defining classes, one that doesn’t use the dialogue provided by the System Browser. Instead, one takes an existing class definition and modifies it to suit the new class. You might consider this the classic method, which is available in various Smalltalk development environments and was even described in the *Smalltalk-80-Bible*.³

As a starting point, we’ll use the class definition for `QuadrEquat`:

```
Smalltalk.QuadEqNs defineClass: #QuadrEquat
  superclass: #(Core.Object)
  indexedType: #none
  private: false
  instanceVariableNames: 'a b c solution'
  classInstanceVariableNames: ''
  imports: ''
  category: ''
```

In order to define the class `Solution`, we’ll replace

- The symbol `#QuadrEquat` with `#Solution` in the first line and

³Goldberg and Robson (1989).

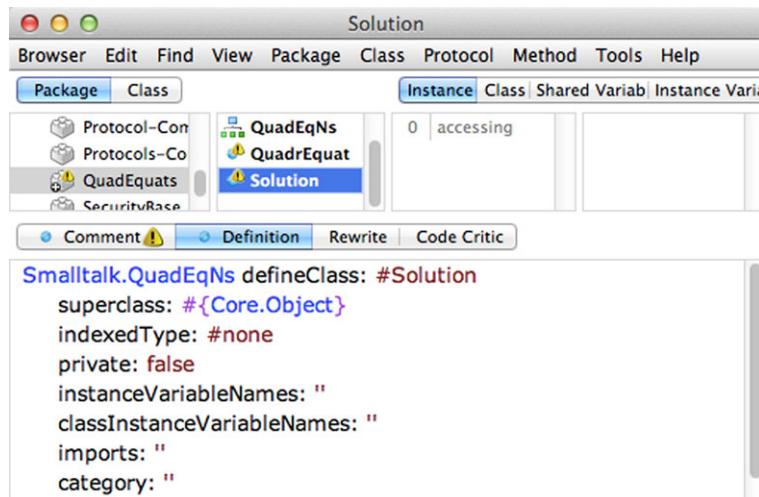


Fig. 8.38 Definition of the class Solution

- The character string 'a b c solution' in the fifth line with an empty character string, because Solution has no instance variables.

Now we have

```

Smalltalk.QuadEqNs defineClass: #Solution
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: ''

```

Clicking **Edit → accept** from the menu bar executes the class definition and creates the class Solution in the same package as the class QuadrEquat (see Fig. 8.38).

To define the class OneSolution, we could proceed as follows:

1. We start with the class Solution and replace the class name with #OneSolution.
2. After superclass:, we write {#Solution}. (The System Browser automatically supplements the namespace QuadEqNs when **accept** is clicked.)
3. After instanceVariableNames:, we write 'x1'.
4. We click on **Edit → accept**.

The definition of the class `OneSolution` is thus:

```
Smalltalk.QuadEqNs defineClass: #OneSolution
    superclass: #(QuadEqNs.Solution)
    indexedType: #none
    private: false
    instanceVariableNames: 'x1 '
    classInstanceVariableNames: ''
    imports: ''
    category: ''
```

In the same way, we can create the other subclasses of `Solution`, whose definitions we show here in the interest of completeness:

```
Smalltalk.QuadEqNs defineClass: #NoSolution
    superclass: #(QuadEqNs.Solution)
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: ''

Smalltalk.QuadEqNs defineClass: #TrivialSolution
    superclass: #(QuadEqNs.Solution)
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: ''

Smalltalk.QuadEqNs defineClass: #TwoSolutions
    superclass: #(QuadEqNs.OneSolution)
    indexedType: #none
    private: false
    instanceVariableNames: 'x2 '
    classInstanceVariableNames: ''
    imports: ''
    category: ''
```

8.5.3 The Solution Methods

At the end of Sect. 8.5.1, we postponed the implementation of the methods

- `QuadrEquat>>solveLinearEquation` and
- `QuadrEquat>>solveQuadraticEquation`

so that we could first develop the structure of the Solution objects. We've done that now, and so we'll proceed with the implementation of these two methods:

```
QuadrEquat>>solveLinearEquation
    "calculates all real solutions for the 'quadratic'
     equation (receiver) for the case of a=0"

^b = 0
ifTrue:
    [c = 0
        ifTrue: [TrivialSolution new]
        ifFalse: [NoSolution new]]
    ifFalse: [OneSolution with: c negated / b]

QuadrEquat>>solveQuadraticEquation
    "calculates all real solutions for the quadratic
     equation (equation) for the case of a~=0"

| radicand root |
radicand := b * b - (4 * a * c).
^radicand = 0
ifTrue: [OneSolution with: b negated / (2 * a)]
ifFalse:
    [radicand > 0
        ifTrue:
            [root := radicand sqrt.
             TwoSolutions
                 solutionOne: (b negated + root)/(2*a)
                 solutionTwo: (b negated - root)/(2*a)]
        ifFalse: [NoSolution new]]
```

Notice that, depending on which alternate solution takes effect, either method responds with an object in the form of the “matching” instance of one of the solution classes

defined in the last section. The calling method `QuadrEquat»solveYourself` (see Sect. 8.5.1) actually uses the set method `solution`: to write this solution object into the instance variable `solution` belonging to a `QuadrEquat` object.

Of course, the new, object-oriented implementation does not change the basic structure of the solution algorithm from Sect. 2.3.3.

Class Methods of the Solution Classes

At this point, we'll expand on the implementation of the methods for creating instances of the classes `OneSolution` and `TwoSolutions`. To create instances of the classes `TrivialSolution` and `NoSolution`, we simply use the message `new`, because their instances do not make use of instance variables.

```
OneSolution class>>with: aNumber
    "creates an instance of the receiver und sets the
     solution to aNumber"

    ^self new x1: aNumber

TwoSolutions class>>solutionOne: aNumber1
    solutionTwo: aNumber2
    "creates an instance of the receiver, that is, a
     solution of a quadratic equation with the real
     solutions aNumber1 and aNumber2"

    ^self new
        x1: aNumber1;
        x2: aNumber2
```

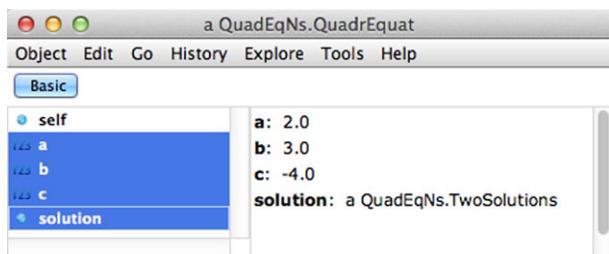
The Method `numberSolutions`

In Sect. 8.5.1, we discussed the fact that the method `numberSolutions` in the class `QuadrEquat` delegates the determination of the number of solutions to the solution object. It's very easy to put this into practice, because a solution object—since it belongs to a class—“knows” how many solutions it represents. An instance of the class `TwoSolutions`, for example, has two solutions. Here are the implementations of the method `numberSolutions` in the subclasses of `Solution`:

```
TwoSolutions>>numberSolutions
    ^2

OneSolution>>numberSolutions
    ^1
```

Fig. 8.39 A “solved” instance of the class `QuadrEquat`



```
NoSolution>>numberSolutions  
^0  
  
TrivialSolution>>numberSolutions  
^3
```

With respect to the number of solutions to quadratic equations, everything that is greater than 2 counts as “infinite.” The number 3 is to be understood in this light, which is the result of the implementation of `numberSolutions` in `TrivialSolution`.

8.5.4 Examples of Applications

We’ve now finished developing all methods for the six classes of object-oriented implementation of the solution of quadratic equations, which means that the expression

```
QuadrEquat a: 2.0 b: 3.0 c: -4.0
```

can be evaluated. Using **Inspect it**, the Inspector shows the `QuadrEquat` instance shown in Fig. 8.39. The equation appears to have two solutions.

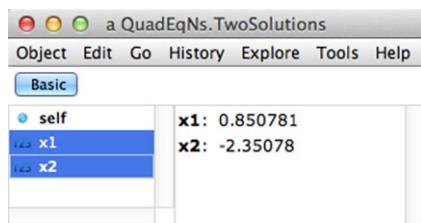
The solution object itself, which one might receive with the expression

```
(QuadrEquat a: 2.0 b: 3.0 c: -4.0) solution
```

is shown in Fig. 8.40 (in the Inspector).

And now it’s finally possible to run the test program shown at the start of Sect. 8.5. Here it is again:

Fig. 8.40 Instance of the class TwoSolutions



```
| eq |
eq := QuadrEquat a: 2.0 b: 3.0 c: -5.0.
eq numberSolutions > 0
    ifTrue:
        [eq numberSolutions = 1
            ifTrue:
                [Transcript cr;
                    show: 'Equation has one solution: ';
                    eq solutionOne printString]
            ifFalse:
                [eq numberSolutions = 2
                    ifTrue:
                        [Transcript cr;
                            show: 'Equation has two solutions: ';
                            eq solutionOne printString,
                            ' and ',
                            eq solutionTwo printString]
                    ifFalse:
                        [Transcript cr; show:
                            'Equation has infinitely many solutions: '
                            ]]]
        ifFalse:
            [Transcript cr;
                show: 'Equation has no real solution']
```

If we use **Do it** to execute it, the following text appears in Transcript:

```
Equation has two solutions: -2.5 and 1.0
```

8.5.5 Use of Inheritance and Polymorphism

The message `numberSolutions`, which is understood by all four concrete solution classes, is a typical example of the use of polymorphism. Depending on the class to which it belongs, each instance of one of these classes reacts differently when it receives the same message.

Since the class `TwoSolutions` is the subclass of `OneSolution`, it inherits the method `numberSolutions` from its superclass. In this case, though, the subclass overwrites the method.

Although the message `solutionOne` is implemented only in the class `OneSolution`, inheritance means that it can also be understood by instances of the class `TwoSolutions`, and that it will make them behave in the same way as instances of `OneSolution`.

8.5.6 Test Programs as Class Methods

Smalltalk expressions that are written and evaluated in a workspace serve primarily to test methods of classes used to create an application program. Workspaces are essentially a temporary medium; they have no formal relationship with the classes of the application and are usually not kept, even though they are saved when the Image is saved. If you store an application in a version-control system, VisualWorks packages are treated as administrative units. But packages do not contain workspaces. The same is true if you use **Package → Publish as Parcel** from the System Browser's menu to save a package to your hard drive, which you might do if you wanted to upload it to a different Image.

Still, it seems unreasonable to just throw away all of the work that you put into creating those test sequences, such as the ones we wrote for testing the methods for the class `QuadrEquat`. In practice, the need to repeat tests always arises, for example, whenever the implementation of a class changes to meet new requirements or to fix bugs. For that reason, it's efficient to define as methods the very Smalltalk sequences that were used to test the methods of a class. Class methods are particularly suited for this task since, as a rule, instances of the classes to be tested must usually be created when the tests are conducted.

As an example, we'll define the contents of the previously developed test workspace as a class method called `ca : cb : cc :` in the class `QuadrEquat`. We can do this in a method protocol called `examples`.

```
QuadrEquat class>>ca: aFloat cb: bFloat cc: cFloat
    "tests the solution methods for a quadratic equation
     with the coefficients aFloat bFloat cFloat"
    "self ca: 2.0 cb: 3.0 cc: -5.0"
```

```

| eq |
eq := QuadrEquat a: aFloat b: bFloat c: cFloat.
eq numberSolutions > 0
    ifTrue:
        [eq numberSolutions = 1
            ifTrue:
                [Transcript cr;
                    show: 'Equation has one solution: ',
                        eq solutionOne printString]
        iffFalse:
            [eq numberSolutions = 2
                ifTrue:
                    [Transcript cr;
                        show: 'Equation has two
                            solutions: ',
                            eq solutionOne
                                printString,
                            ' and ',
                            eq solutionTwo
                                printString]
                iffFalse:
                    [Transcript cr; show:
                        'Equation has infinitely
                            many solutions'
                    ]]]
    iffFalse:
        [Transcript cr;
            show: 'Equation has no real solution'].
^eq

```

The body of the method corresponds in large part to the content of the test program. In addition, it produces the instance of `QuadrEquat` stored in the temporary variable `eq` as a result.

When you want to run a test, you only need to enter the expression

```
QuadrEquat ca: 2.0 cb: 3.0 cc: -5.0
```

in a workspace. Evaluating the expression with **Do it** invokes the display shown above in the Transcript. Since the method produces the `QuadrEquat` object as a result, you can examine it in the Inspector when you evaluate it with **Inspect it**.

Instead of using the workspace to execute the method, you can also go into the System Browser and mark the comment

```
self ca: 2.0 cb: 3.0 cc: -5.0
```

directly in the code, and then use **Do it** to evaluate it.

Systematic Testing

The advantages of programming previously developed test programs as class methods are obvious:

- They are saved in the package as components of a class definition.
- They can be repeated without much effort.

It's necessary to point out, though, that the test programs we've been using so far are not real tests as they are understood in software engineering. That's because the definition of a test includes not just the writing of a test method that activates the methods you want to test; it must also include a description of the expected results. Only when the test includes both parts can you determine whether the actual results of a concrete test run accord with the expected results. Writing that kind of test is a time-consuming task, which in software projects often falls victim to the ever present time pressure. Therefore it's all the more important that one know—and consistently use—the means that development environments make available to support programmers as they develop and execute tests. Chapter 15 looks at this topic in depth.

8.5.7 Error Management

An expression like

```
(QuadrEquat ca: 0 cb: 0 cc: 1) solutionTwo
```

makes no sense from a technical point of view, because the equation obviously has no solution. From a programming point of view, though, it's certainly worth asking yourself what should happen in this case. Without further actions, the program will suffer an exception and issue the error message:

Unhandled exception: Message not understood: #x2

You could be content with that, because you certainly can't expect a solution from the `QuadrEquat` object shown above. After all, you can use the message `numberSolutions` to test beforehand whether any solutions exists. Nevertheless, it seems reasonable in this case to avoid the message-not-understood exception and to create an individualised program exception combined with a specific error message.

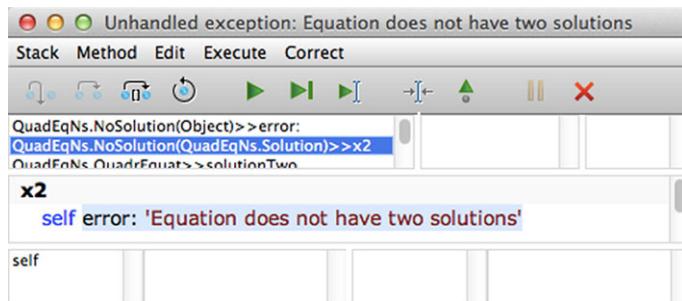


Fig. 8.41 Exception caused by Solution»x2

There are many ways to handle this in Smalltalk and in VisualWorks. The topic of *exception handling* is both very complex and strongly dependent on the programming language. Cincom Systems (2014b) handles this topic in detail. For our example, we'll provide a very simple process for ensuring that all messages for which the class `QuadrEquat` provides a method either provide a correct result or else cause the program to abort with a specially generated error message.

The error message shown above suggests that the message `solutionTwo` is not understood by an instance of the case `NoSolution`. A similar situation would arise if the same message were sent to an instance of `OneSolution`. In order to create a program exception with its own error message, we define a method called `solutionTwo` in the superclass `Solution` that does exactly that:

```
Solution>>x2
    self error: 'Equation does not have two solutions'
```

Every object understands the message `error:`, which causes an exception and displays an error message displaying the character string passed as an argument.

When we evaluate the expression

```
(QuadrEquat ca: 0 cb: 0 cc: 1) solutionTwo
```

the debugger opens and displays the error message shown in Fig. 8.41.

For the sake of consistency, one should use this model to create the following method:

```
Solution>>x1
    self error: 'Equation does not have a solution'
```

With both methods in place, any attempt to obtain a solution from a `NoSolution` instance would always produce an appropriate error message. But since the method is implemented in the class `Solution`, the same thing would happen if we evaluated the expression

```
(QuadrEquat ca: 0 cb: 0 cc: 0) solutionOne
```

as well as with `solutionTwo`. The statement that the equation has no solution is, of course, mathematically false. The reader should consider how this problem can be resolved.

When you evaluate Smalltalk expressions, you may receive error messages when either the external form of the expression or the way that it's applied is not correct. The type of error message depends on what point the processing of the expression had reached when the error occurred. We can make a crude distinction between errors discovered by the compiler as it attempts to translate the Smalltalk expression into the internal byte code and exceptions that are caused by incorrect application of methods.

The following sections explain the following types of errors:

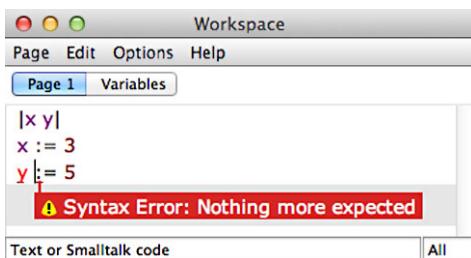
- Syntax errors
- Unknown variables
- Unknown message selectors
- Exceptions (runtime errors)

At the end of the chapter, we'll explain how to use the debugger.

9.1 Syntax Errors

When the compiler attempts to translate a Smalltalk expression into internal byte code, it first has to check whether the expression was written according to the syntactic rules of the Smalltalk language. Among the most common syntax errors are incorrectly placed periods, semicolons, parentheses or square brackets. In these cases, the compiler writes the error message directly into the program text below the location of the error. Figure 9.1 shows an example. The error message *Nothing more expected* often indicates two Smalltalk instructions that are missing a period separating them. In the example, the period is missing in the second line. The error message occurs in this case in the third line because the compiler assumes that the expression in the second line continues into the third, but then encounters a misplaced assignment symbol.

Fig. 9.1 Display of a syntax error in a program text



Additional error messages of this kind include:

- *Period or right bracket expected*—usually indicates an error in the structure of square brackets.
- *Right parenthesis expected*—indicates a missing right parenthesis

9.2 Unknown Variables

In addition to pure syntactic analysis, the compiler also checks whether all the variables are known that appear in the expressions it will translate. If it encounters an undeclared variable (see Fig. 9.2), the error message contains links which allow you to declare the missing variable as *Workspace Variable*, *Temporary Variable* or leave it undeclared.

If you've simply forgotten to declare a temporary variable, just click **Temporary** and the missing declaration will be added to the program text. If you click **Undeclared List**, the variable will be declared as a Shared Variable in the namespace Undeclared (see Fig. 9.3). Normally, you should avoid using this option. You can also hit the **Esc** key and correct the program text yourself.

When you declare a new variable, it can sometimes happen that its name conflicts with that of one that's already been defined. For example, if you declare a local variable in a method that has the same name as an instance variable, the compiler will recognise the conflict and report it, as shown in Fig. 9.4.

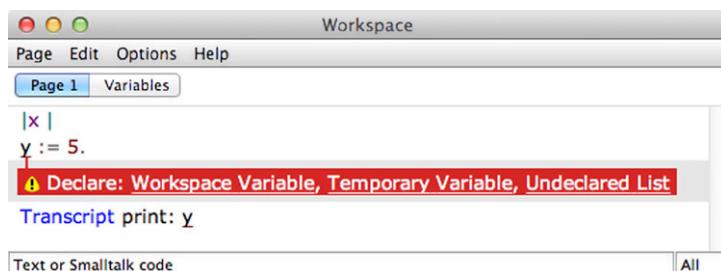


Fig. 9.2 The variable y has not been declared

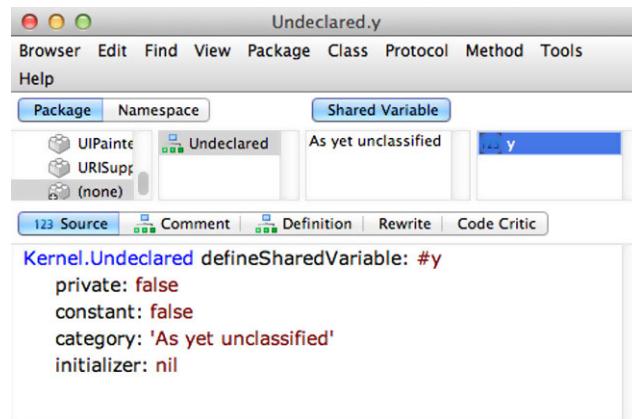


Fig. 9.3 The shared variable `y` in the namespace `Undeclared`

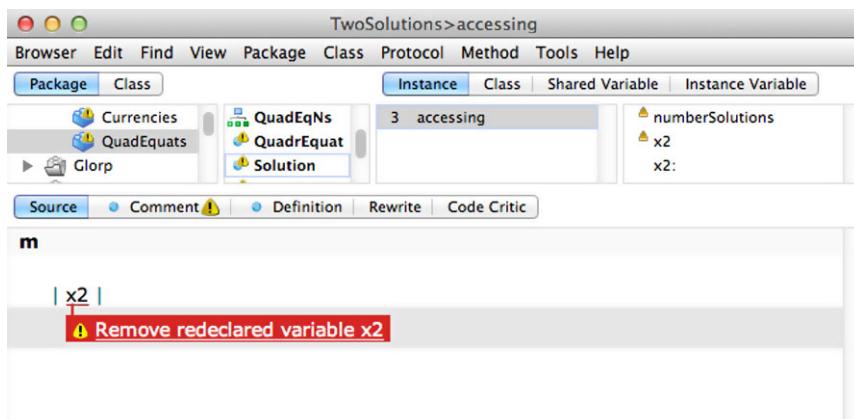


Fig. 9.4 Error message for conflicting variable declarations

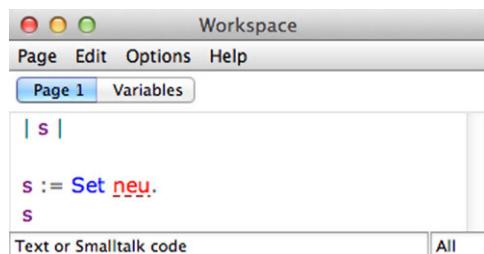
Each variable identifier may be declared only once within its scope (see Sect. 3.3). In this context, note that different kinds of variables may have differing but possibly overlapping scopes. For example, the scope of an instance variable extends over all instance methods in its class, thus overlapping with the local variables defined in these methods.

9.3 Unknown Message Selectors

The compiler also recognises whether a Smalltalk expression contains message identifiers for which no method definition exists in the entire Image. In that case, the wrong message selector is underlined by a dashed line (see Fig. 9.5).

Fig. 9.5 Unknown message

selector neu



You can also ignore the missing method and let the translation process proceed. This is a popular option if a new method is being translated that contains message identifiers for additional methods that still need to be defined. No later than when you activate the method though, methods must exist for all of the message identifiers you use in it. Otherwise, you'll encounter an exception (see Sect. 9.4).

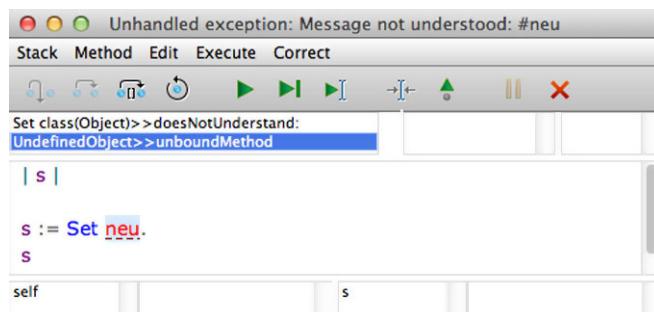
9.4 Exceptions

Exceptions are error situations that are discovered while a Smalltalk program is being processed by the VM, and which then lead to a program abort or to an interruption of the program when the debugger is activated.

Among the most common exceptions to occur during program development is the attempt to activate a method that doesn't exist. If you ignore the wrong message selector shown in Fig. 9.5 and select **Do it** from the **Edit** menu, the program will be translated in the workspace and handed off to the VM for processing. The VM, though, will report that the class `Set` does not understand the message `neu` (see Fig. 9.6).

Another cause for the occurrence of the exception *Message not understood* might not be that one has made a mistake in choosing a message selector—as was the case in the previous example—but rather that a message has been sent to an unsuitable object. This might happen if one sent a string message to a variable that is bound to `nil`.

Other typical exceptions include:

**Fig. 9.6** The object Set does not know the message neu

- Division by 0 (Error message: *Can't divide a number by 0*)
- Attempt to access a container (such as `Array`) with an invalid index (Error message: *Subscript out of bounds*)

If the error message in the title of the debugger window provides sufficient information about the reason for the error, you can usually end the program execution by closing the window or by clicking **Execute → Terminate** from the menu bar. There are only a few instances where it makes sense to click **Execute → Run** to let the program continue to run.

Section 4.1 explained how to use the debugger to pinpoint the place in the program that led to the exception. The following section shows how to use the debugger step-by-step to execute a Smalltalk program.

9.5 Debugging Methods

An important use for the debugger is to let it execute a Smalltalk program step-by-step, that is, message-by-message. This is something you might do in order to track down an error when a program does not produce the desired result. When you run a program step-by-step, you can inspect the variables in the intermediate state of the program each time you send a message.

This also lets you observe the processing of a program. For this purpose, we'll now use the debugger to step through the program for solving a quadratic equation that we created in Sect. 8.5.

As a starting point, we'll use the test method defined in Sect. 8.5.6 and evaluate the expression

```
QuadrEquat ca: 2.0 cb: 3.0 cc: -5.0.
```

in the workspace; this time we'll click on **Debug it** in the menu bar. At that point, the debugger window shown in Fig. 9.7 appears.

Back in Sect. 4.1 (and in Fig. 4.3), we explained the basic structure of the debugger window. In Field 1, the so-called `unboundMethod` is highlighted, which is the method in the workspace that is in the process of being executed. Its method text appears in Field 2. We won't describe the use of the so-called stack Inspectors in Fields 7 and 8.

The next message to be sent is always highlighted in Field 2. Three buttons above Field 1 are available for sending the next message; the **Execute** menu contains equivalent commands:

- **Step** executes the method belonging to the message in a debugger step.
- **Step into** causes the debugger to process the method belonging to the message in a step-by-step fashion.

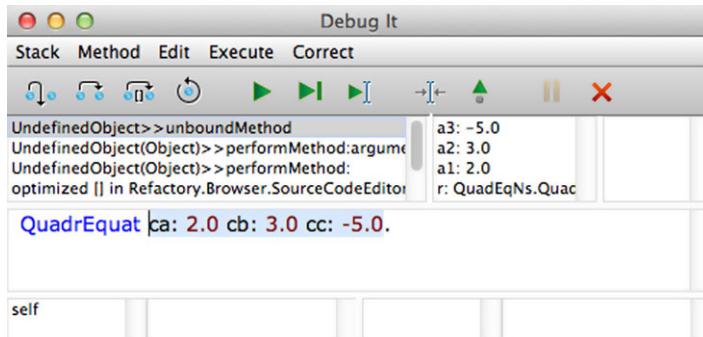


Fig. 9.7 Starting the debugger with a program from the workspace

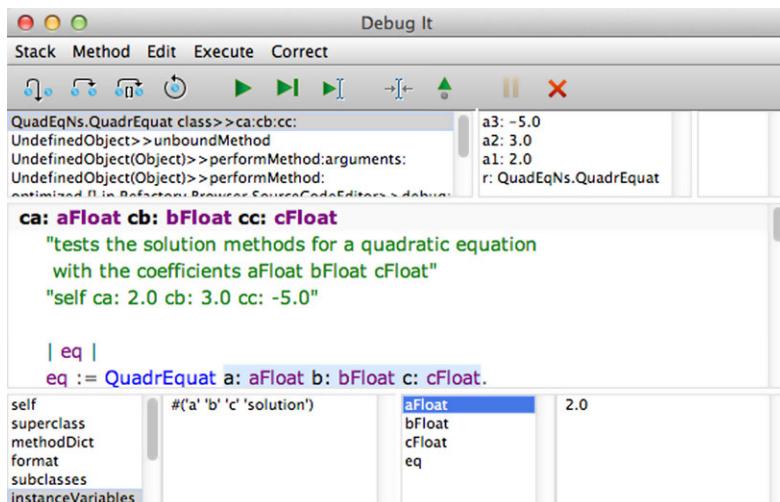


Fig. 9.8 The beginning of the class method ca:cb:cc:

- **Step over** performs like **Step into**, except that any blocks that occur are processed together in one step. This can be especially useful when you do not want to perform step-by-step processing of repetition structures.

If you were to click the **Step** button in the situation shown in Fig. 9.7, the `unboundMethod` would be processed in a single step, and in this case, the processing of the program in the workspace would be over and done with.

In contrast, if you were to click **Step into**, the debugger would jump to the beginning of the method `ca : cb : cc :` and execute it. Starting from the situation shown in Fig. 9.7, clicking the **Step into** button leads to the debugger window shown in Fig. 9.8.

Above the `unboundMethod` in Field 1 the line

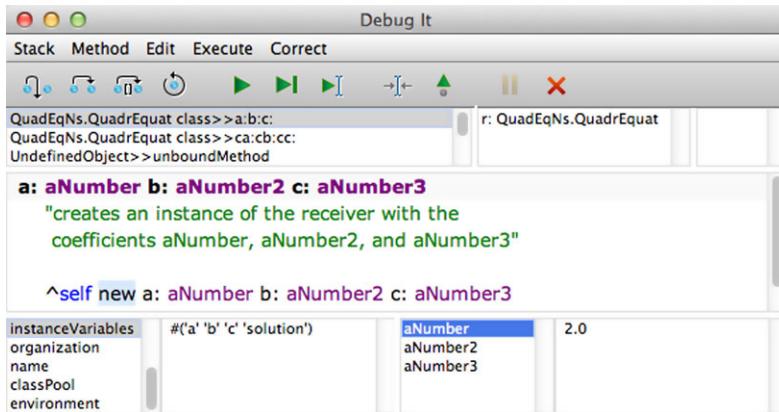


Fig. 9.9 After executing `ca: 2.0 cb: 3.0 cc: -5.0`

```
QuadrEquat class>>ca:cb:cc:
```

appears. The word `class` after `QuadrEquat` indicates that the method `ca:cb:cc:` is a class method. The text of this method appears now in Field 2, where the selected message `a: aFloat b: bFloat c: cFloat` is now the one that will be sent next.

You can see in Field 5 that it's not just a method's temporary variables—the variable `g1` in this case—that are displayed, but also placeholders for the message arguments that were used in the first line of the calling pattern. To illustrate, in Fig. 9.8, the placeholder `aFloat` is highlighted (with its value in Field 6).

At this point, it's necessary to make a few basic comments about Fields 3 and 4 of the debugger window. Field 3 shows the instance variables of the object for which a method is currently being executed, and Field 4 shows the value of the instance variables highlighted in Field 3. In our example, though, Field 3 does not display the instance variables that were introduced in the definition of the class `QuadrEquat`. That is because the object for which the class method `ca:cb:cc:` is being carried out is not an instance of the class but rather the class itself. Since classes are objects, they are also instances of a class, that is, of whichever so-called *metaclass* they belong to. These metaclasses also establish instance variables for their instances, the classes. Section 11.3 explains the significance of metaclasses in greater detail. Simplifying a little, we can say that the instance variables that were defined in the metaclass of `QuadrEquat` appear in Field 3. In Fig. 9.8, we see that the variable `instanceVariables` has been selected. As a value, Field 4 displays an array with the names of the instance variables of the class `QuadrEquat` as a character string.

Now let's continue with the execution of the class method `ca:cb:cc::`. If we again click **Step into**, the debugger shows the situation displayed in Fig. 9.9.

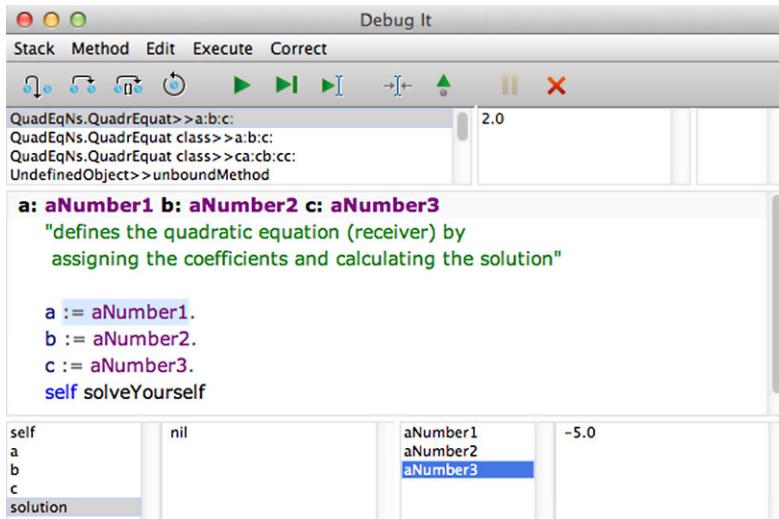


Fig. 9.10 The activation of a:b:c: has occurred

In Field 2, we can see that the next message to be executed is new, which is being sent to self, that is, to the class QuadrEquat. The method to be executed is from the class Object. This can't be executed on a step-by-step basis because it's directly implemented within the VM. That means that, in this case, **Step** and **Step into** perform the same.

If we first execute **Step** and then **Step into**, first a new instance of QuadrEquat will be created, after which the message a: aFloat b: bFloat c: cFloat will be sent to the instance. You can see the result in Fig. 9.10. In the first line of Field 1, the entry

```
QuadrEquat>>a:b:c:
```

indicates that it's now the instance method a:b:c: that's being executed and not the class method with the same name, which has now slipped into the second line. In Field 3, we also now see the instance variables of the QuadrEquat object, which is the receiver of the message that led to the activation of the instance method a:b:c:.

Figure 9.11 shows the state of the processing of the method after the three assignments have been executed. We can see that the instance variable c has received the value -5.0.

In Field 2, the message solveYourself has been selected. Clicking **Step into** produces the screen shown in Fig. 9.12. You can see that at first the message “= 0” was sent to a. Since this instance variable is not equal to 0, it appears that the message solveQuadraticEquation must be sent. You can convince yourself that is so by clicking **Step** twice. Figure 9.13 shows the result.

This time we will not **Step into** the method solveQuadraticEquation, but will instead click **Step** twice. The result shown in Fig. 9.14 indicates that an instance of the

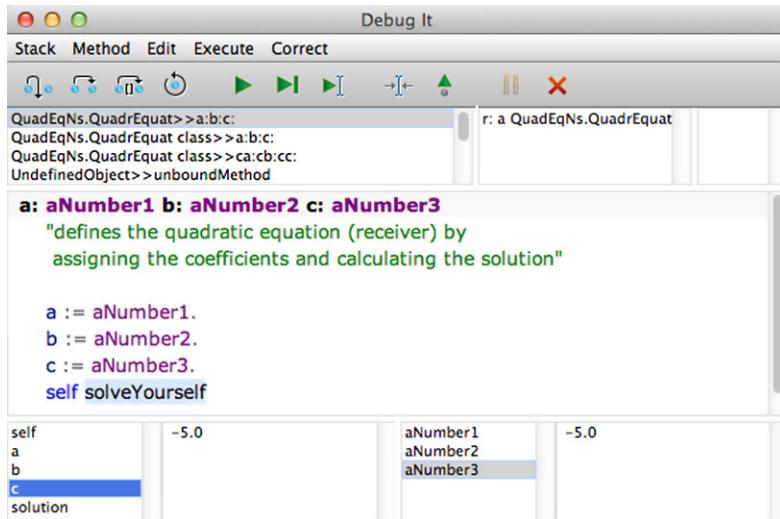


Fig. 9.11 The instance method `a:b:c:` before activating `solveYourself`

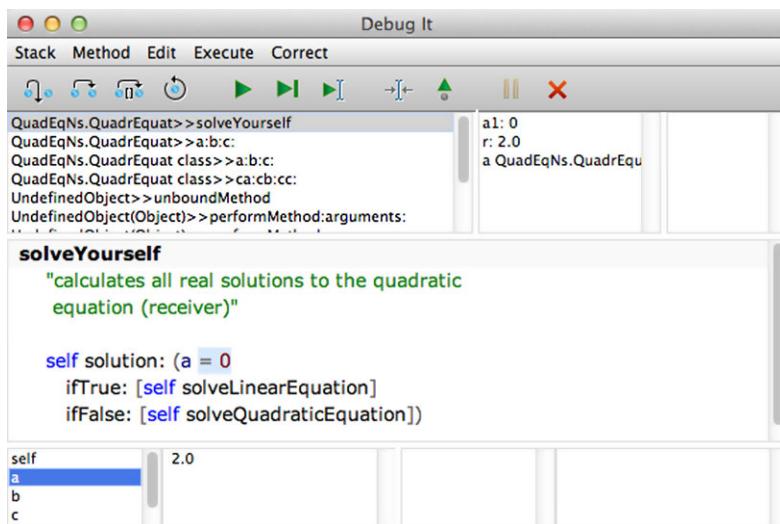


Fig. 9.12 Activating the method `solveYourself`

class `TwoSolutions` was assigned to the instance variable `solution` (with the assistance of the set method `solution:`).

The method `solveYourself` has now been processed. Clicking **Step** again returns us to the instance method `a:b:c:`, which is also finished, so that an additional click on **Step** leads us again to the class method `a:b:c:`, as shown in Fig. 9.15. The next step simply executes the return operator, which returns the (solved) quadratic equation to the class

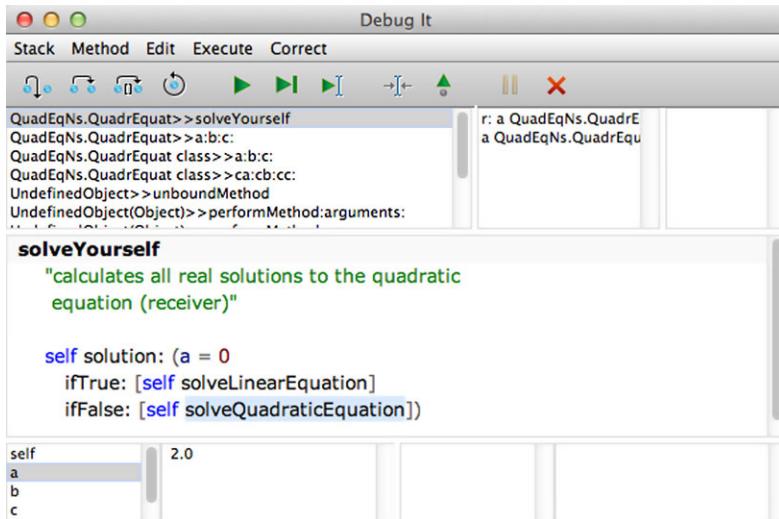


Fig. 9.13 Executing the `ifFalse:` block because $a \neq 0$

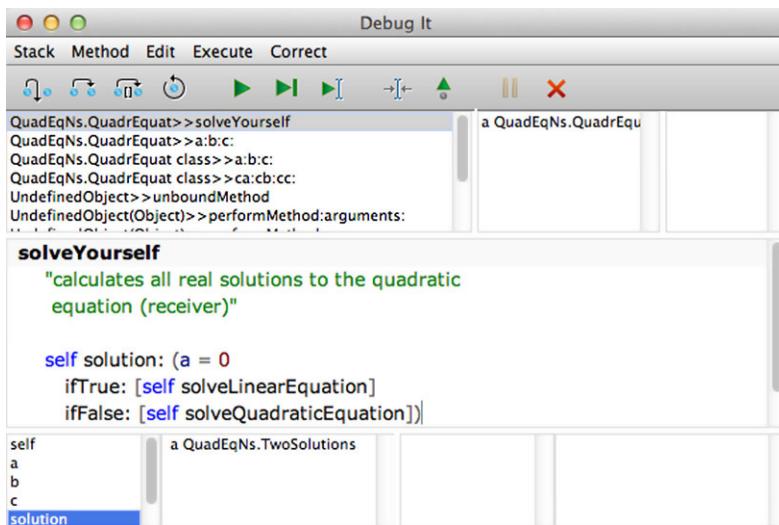


Fig. 9.14 An instance of `TwoSolutions` was created

method `ca : cb : cc :`. When we now click **Step** two more times, Fig. 9.16 shows that an instance of the class `QuadrEquat` has been assigned to the local variable `eq`.

Now you can proceed through the class method by clicking **Step** several times, observing as you do so that the result is output in Transcript. And finally, the last expression, `^ eq`, leads to a return to the `unboundMethod`, which was the starting point of the exercise. And that finishes the step-by-step execution of the sample program.

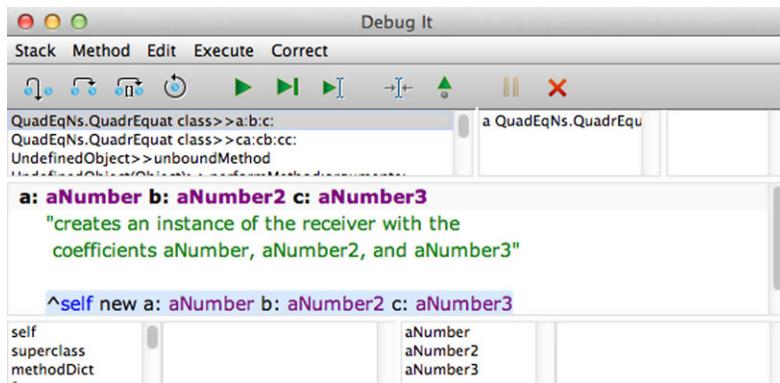


Fig. 9.15 Class method `a:b:c:` is complete

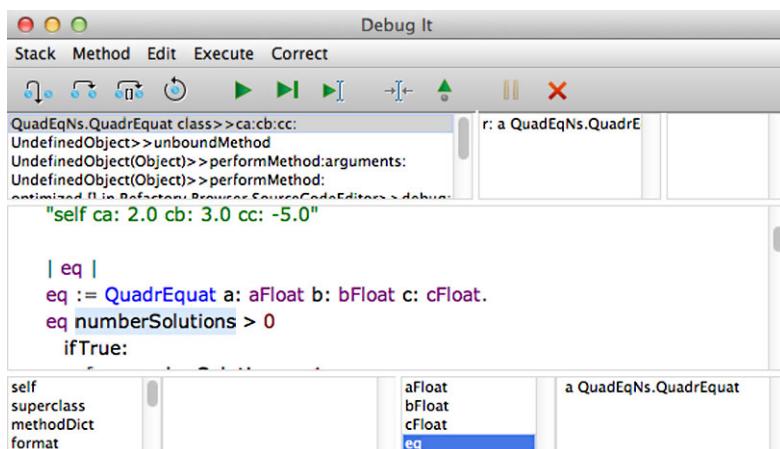


Fig. 9.16 `eq` indicates a solved quadratic equation

Other Capabilities of the VisualWorks Debugger

Besides the step-by-step execution of methods, the VisualWorks debugger has other capabilities that can be useful when tracking down program errors. At this point, we'll just describe them briefly. For more extensive information, see the original VisualWorks documentation (Cincom Systems 2014b).

Sometimes it can be somewhat bothersome to use the step-by-step method to track down an error, perhaps because one is interested only in the step-by-step execution of a single method. In such cases, it's advantageous to be able to start the debugger exactly at the point when that method is activated. You can do this by setting a so-called *breakpoint*. After using the System Browser to set breakpoints in one or more methods, start the program as you normally would, that is, not using **Debug it**. Whenever one of the methods that you provided with a breakpoint is executed, it activates the debugger. You can then

decide whether to run through the method in step-by-step mode or to let the program proceed normally. The program then continues to run without interruption unless during the process it again encounters a breakpoint.

Another aid in the debugging process are so-called *watchpoints*. When a watchpoint occurs during program processing, a special window opens without interrupting the processing. There are watchpoints, for example variables and temporary variables, and the watchpoints can be used to monitor the values of the variables. The text message that is created for this purpose arises when the message `debugString` is sent to the object that is bound to the variable. If no suitable method is available in the pertinent class, the method `printString` can be used.

You can place both watchpoints and breakpoints wherever you like within a method. For watchpoints, the text message is displayed in the observation window whenever the program execution passes the relevant place.

To place a breakpoint, place the cursor in Field 5 of the System Browser at the point where you want to interrupt the program. Then select **Edit → Insert Breakpoint** from the menu. To set a watchpoint, use the command **Edit → Insert Probe...**.

The problem frequently occurs in programming that one must manage quantities or collections of objects. A business application can deal with many customers, vendors, employees or products. One has to be able to address the quantity of all products, the product catalogue, as though it were a unit. In object-oriented languages, that might mean that there's an object called `productCatalogue` that contains the individual product objects as components.

Up until now, we've only encountered objects that can contain any number of objects as components in the form of arrays. Components of an object are usually stored in its instance variables. It's impossible, though, to store a quantity of components in individually named instance variables when the size of that quantity can't be predicted. And when we consider the class definition of the class `Array` (see Fig. 10.1), we see that there are no definitions at all for instance variables. Instead, after the keyword `indexedType:`, we find the symbol `#objects`. For other classes that have named instance variables, we would find in this position the symbol `#none`.

This establishes that an object in the class `Array` can have any number of instance variables—the number to be fixed during creation; these instance variables are not named, though, but are numbered beginning at 1.

An `Array` instance created with `Array new: 10` has 10 instance variables identified by the numbers (indexes) 1 to 10.

In addition to the class `Array`, there are many other so-called container classes. They are all subclasses of a common abstract superclass `Collection`. Figure 10.2 shows a section of the class hierarchy including the collection classes that are most important for application programming.

The internal storage of components for instances of all of these collection classes occurs according to the principles explained for arrays, but their components cannot always be accessed externally by supplying an index. Based on that fact, we can divide the container classes into two categories:

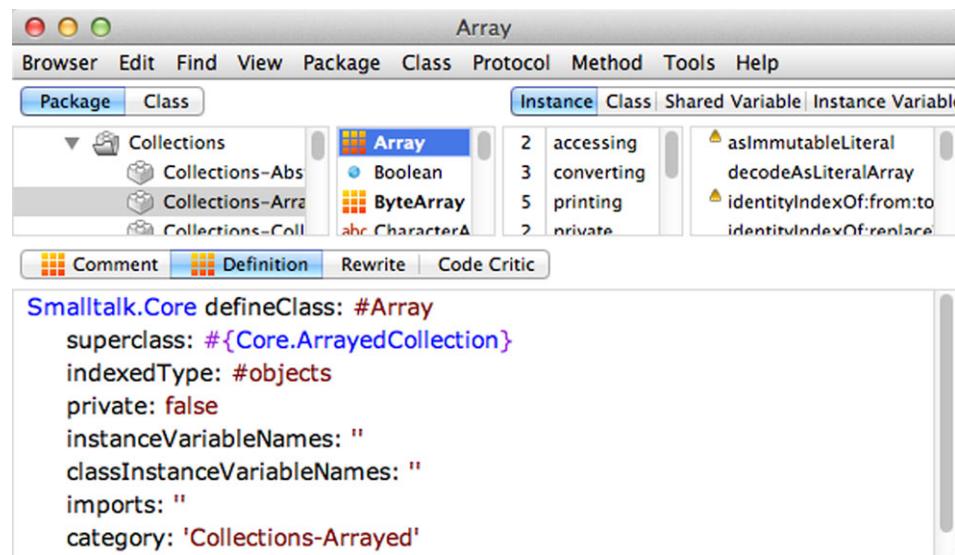


Fig. 10.1 Definition of the class `Array`

Fig. 10.2 Hierarchy of important collection classes

```

Collection ()
Bag ('contents')
SequenceableCollection ()
    ArrayedCollection ()
        Array ()
        CharacterArray ()
        String ()
        Symbol ()
        Text ('string' 'runs')
        List ('dependents' 'collection' 'limit' 'collectionSize')
        Interval ('start' 'stop' 'step')
        OrderedCollection ('firstIndex' 'lastIndex')
        SortedCollection ('sortBlock')
Set ('tally')
Dictionary ()

```

1. Unordered collections, which essentially consist of the classes Bag, Set and Dictionary.
2. Ordered collections, which essentially consist of subclasses of the abstract class SequenceableCollection.

It is only for ordered collections that you can directly access individual components using the format

```

aContainer at: anInteger
aContainer at: anInteger put: oneObject

```

The message `at:` reads a component, while `at:put:` writes.

One further peculiarity of a very few container classes consists of the fact that their components may only be objects of particular classes. Among these are the classes `String` and `Symbol`, whose components are instances of `Character`.

We'll explain more as we examine the two categories described above.

10.1 Unordered Collections

10.1.1 The Class Set

Instances of the class `Set` are collections that basically behave like mathematical sets. Above all, this means that Sets may not contain duplicates and that the elements are not ordered.

Creating Sets

You can create an empty set using the expression

```
Set new
```

The following examples illustrate additional messages for creating `Set` objects.

Adding Elements to a Set

Use the message `add:` to add objects to a set. When we use Inspector to examine the result of the following series of messages, we see the `Set` object shown in Fig. 10.3.

```
| set |
set := Set new.
set add: 5.
set add: 'five'.
set add: $5.
set add: 5.
set
```

We can see that the set contains three elements. The repeated addition of 5 in the second-to-last line has no effect.

If we select the tab **Basic** in Inspector rather than **Elements** (see Fig. 10.4), we gain some insight into the way instances of the class `Set` are actually stored. We can see that sets also contain an instance variable called `tally`, which contains a count of the “valid” elements in the set. In addition, we see that the `Set` instance we are considering currently has seven components, of which four display the value `nil`. Stated somewhat simply,

Fig. 10.3 A set with three elements

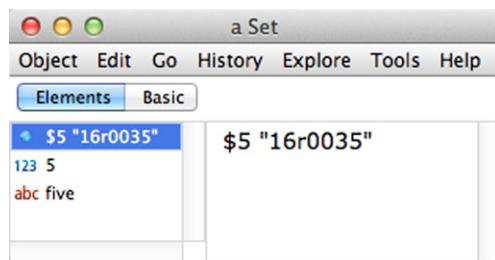
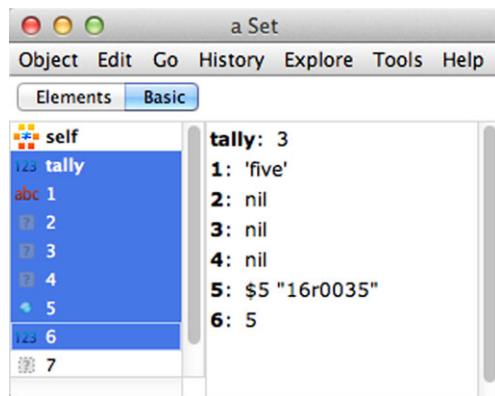


Fig. 10.4 The “internal view” of the Set object from Fig. 10.3



these “excess” components can be said to have been created as an efficiency measure in preparation for accepting additional elements.

Testing Characteristics of a Set

There are two messages available to determine whether a set contains a specific element. The message

```
aSet includes: anElement
```

produces `true` if `aSet` contains `anElement`; otherwise it produces `false`. The message

```
aSet occurrencesOf: anElement
```

produces the frequency with which `anElement` occurs in `aSet`. For sets, of course, 0 and 1 are the only possible results. These two messages, though, are understood by instances of all collection classes.

We can also check whether a set is empty (`isEmpty`, `notEmpty`). The expression

```
(Set with: (Set new)) isEmpty
```

produces `false`. The message `with:` allows you to create a set containing the element sent as an argument. A set that contains as its only element the empty set is nevertheless not itself empty.

You can use the message `size` to query every collection and therefore also a `Set` object about the number of its components.

Removing Elements from a Set

You can, of course, also remove elements from a set. One option is to use the message `remove:;`; this can lead to an exception, though, if the element you want to remove is not part of the set. To avoid exceptions, you can instead use the message `remove:ifAbsent:,` which expects, after the second keyword, a block that will be evaluated precisely when the element you asked to remove is not part of the set.

In addition, you can use the binary message “`-`” to determine the difference between two sets:

```
| set set2 |
"Create a set with the elements 1, 2, 3, 4"
set1 := Set withAll: #(1 2 3 4).      "Print it:
                                         Set (1 2 3 4)"
"Create a second set:"
set2 := Set withAll: #(4 5 6 7).      "Print it:
                                         Set (7 4 5 6)"
"SetDifference:"
set1 - set2 "Print it: Set (1 2 3)"
```

It's interesting that set difference is the only operation in set algebra that has a method. Other set operations, such as intersections and unions, need to be supplemented when necessary.

Processing the Elements of a Set

In Sect. 4.2.3, we introduced the standard message `do:` for processing the elements in an array. This message can be used on instances of all collection classes.

For example, you can use the expression

```
(Set withAll: #(1 2 3 4))
    do: [:elem | Transcript show: elem factorial
                           printString; cr]
```

Table 10.1 Additional messages for processing the elements of a set

Message Pattern	Meaning
collect : aBlock	Evaluates aBlock for each element. Responds with a collection of the same size with the result of the evaluation of the block for each element
select : aBlock	Evaluates aBlock for each element. Responds with a collection that contains only those elements for which aBlock evaluates to true. (The collection may be empty)
reject : aBlock	Evaluates aBlock for each element. Responds with a collection that contains only those elements for which aBlock evaluates to false. (The collection may be empty)
detect : aBlock	Evaluates aBlock for each element. Responds with the first element for which aBlock evaluates to true. In the alternative, you can use the message detect : ifNone : . In that case, the second argument is a block that is evaluated when no elements of the receiver agree with the criteria supplied in the first argument. Otherwise, detect: responds with an error
inject : anObject into : aBlock	aBlock must have two block parameters. In the first, a value is accumulated that results from the evaluation of the argument aBlock with the current value of the receiver (2nd block parameter). The starting value is the value of the argument anObject

to print to Transcript the factorials of all numbers between 1 and 4.

It's a characteristic of Smalltalk that it has a series of additional messages available for special applications involving the processing of collections; many of them can also be used on sets. Table 10.1¹ summarises the way these messages operate. When they are sent to a Set object, the first three messages produce a Set object as a result. Table 10.2 shows examples of various applications for these messages.

10.1.2 The Class Bag

The characteristics of bags and set differ in only one way. The same object can occur more than once in a bag. In other words, evaluating the following sequence of messages

```
| sack |
sack := Bag new.
sack add: 5.
sack add: 'five'.
sack add: $5.
```

¹This presentation is based on Hopkins and Horan (1995).

Table 10.2 Application examples for the messages in Table 10.1

Expression	Result
(Set with:1 with:3 with:4) collect[:each each factorial]	Set (1 24 6)
(Set with:1 with:3 with:4) collect[:each each >= 3]	Set (false true)
(Set with:1 with:3 with:4) select[:each each >= 3]	Set (3 4)
(Set with:1 with:3 with:4) select[:each each > 4]	Set ()
(Set with:1 with:3 with:4) reject[:each each >= 3]	(Set (1))
(Set with:1 with:3 with:4) detect[:each each >= 3]	3
(Set with:1 with:3 with:4) detect[:each each > 4] ifNone: ['Not found']	'Not found'
(Set with:1 with:3 with:4) inject: 0 into: [:sum:each sum + each]	8

```
sack add: 5.
sack size
```

yields 4 as a result, because the object 5 occurs twice in the Bag object.

Otherwise, everything that we've said about sets can be applied—with the appropriate changes—to bags.

10.1.3 The Class Dictionary

In Fig. 10.2, you can see that Dictionary is a subclass of Set. In other words, Dictionaries are sets with special properties. One peculiarity is that it is a homogeneous collection. That means that all elements of a Dictionary object are instances of the same class. A Dictionary is a set of instances of the class Association. Association objects have two instance variables, key and value, which represent so-called key-value pairs. A further special property is that instances of the class Dictionary—in contrast to other unordered collections—understand the messages at: and at:put:; they understand them, though, in a somewhat modified way.

You can use dictionaries to create tables of any kind. Although a dictionary may only contain associations as its elements, the components of an association may be any kind of object. In order to put an entry into a dictionary, the at:put: message is used in the following manner:

```
aDictionary at: aKey put: aValue
```

Fig. 10.5 A Dictionary with three entries



Fig. 10.6 The internal storage format for a Dictionary object



This message adds an association with the key `aKey` and the value `aValue` to the dictionary `aDictionary`. As we said, the objects `aKey` and `aValue` can be instances of any class. Frequently though, character strings or symbols are used for the keys.

As an example, let's look at the creation of a primitive dictionary:

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic
```

In this case, we have created a dictionary with three associations. Figure 10.5 shows the result in Inspector. With the tab **Elements** selected, the image shows the keys on the left, and on the right, the words assigned to them. If we select the tab **Basic** instead, we gain insight into the internal storage format of a Dictionary object (see Fig. 10.6). The instance variable `tally`, inherited from the class `Set`, shows the number of valid entries. The elements of this set are associations, which are shown on the right in the format

Fig. 10.7 An instance of the class Association

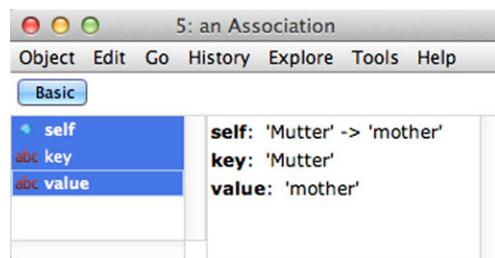
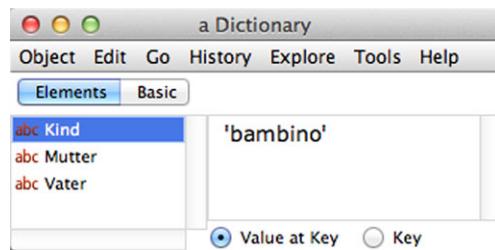


Fig. 10.8 The association 'Kind' -> 'child' has been overwritten



Key->Value

You can convince yourself that these entries are instances of the class `Association` by selecting an entry and then clicking **Go** → **Dive** → **yourself** from the menu. An Inspector window like the one in Fig. 10.7 appears at that point.

If you add to a dictionary an association with a key that already exists, the new value overwrites the old one. A key may exist only once in a dictionary. This is a consequence of the fact that Dictionaries are sets. The evaluation of the series of messages

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic at: 'Kind' put: 'bambino'.
dic
```

produces the Dictionary object shown in Fig. 10.8.

Adding Associations

In the earlier material, we've seen examples for applying the message `at:put:`. An alternative way to add an association to a dictionary is to use the message `add:`, which we

already know from using it with sets. In this case, the argument must be an association. One option for creating an association is to use the binary message “`->`”:

```
( 'Sohn' -> 'son') class "Print it: Association"
```

We can use

```
dic add: 'Sohn' -> 'son'
```

to add this association to our dictionary.

Removing Associations

You must use a special message to remove associations from a dictionary. You cannot use the `remove:` message for sets. Instead, the messages `removeKey:` and `removeKey:ifAbsent:` are available. In other words, the association you want to remove is identified by supplying the key. The second message allows you to indicate a block that will be evaluated in case no association for the indicated key exists in the dictionary.

Accessing Dictionaries

The simplest method for accessing an association in a dictionary is to use the message `at:`, with the desired key as an argument:

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic at: 'Mutter' "Print it: 'mother'"
```

As you can see, `at:` yields the value belonging to the key. If the key does not exist an error occurs. As an alternative, you can avoid an exception by using the message `at:ifAbsent:`, which again expects a block after the second keyword.

To receive as a response the `Association` object belonging to a key, you can use the message `associationAt:`; this message also has a variation with the second keyword `ifAbsent::`.

The message `keys` yields the keys that appear in a dictionary in the form of a `Set` object; the message `values` yields the dictionary's values in an `OrderedCollection` collection (see Sect. 10.2).

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic keys. "Print it: Set ('Mutter' 'Vater' 'Kind')"
dic values
"Print it:
OrderedCollection('father' 'mother' 'child')"
```

Testing a Dictionary's Properties

You can use the message `includes:` to test a dictionary for the presence of a value, and `includesKey:` for the presence of a key:

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic includesKey: 'Mutter' "Print it: true"
dic includes: 'father' "Print it: true"
```

Looping Through the Elements in a Dictionary

As is true for all collections, you can use the message `do:` on instances of the class `Dictionary`. In that case, though, remember that the message loops only through the values and not the associations. The message sequence

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic do: [:elem | Transcript cr; show: elem printString]
```

creates the output shown in Fig. 10.9 in Transcript. Think of the following analogy as an explanation: You use the message `at:` to access the components of a dictionary and receive as a response the value stored at the indicated key. When you access an array, as the argument of the message `at:,` you supply the index of the desired element. You can

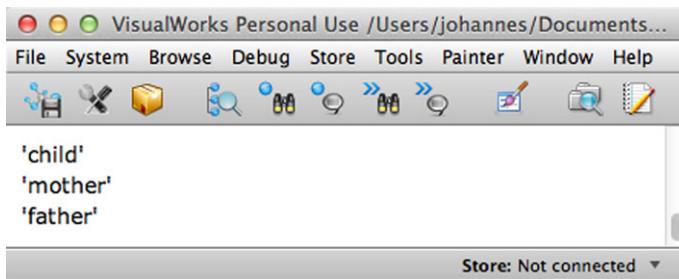


Fig. 10.9 The do : message loops through the values of a dictionary

imagine that key and value pairs are stored in an array, and that the indexes assume the role of the keys. When you use the do : message in an array, you receive the array elements (the values) but not the index-value pairs.

The way that the do : message behaves also affects other messages implemented using do : that are used for looping through collections. This applies to all of the messages shown in Table 10.1. For example, the message sequence

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
dic collect: [:elem | elem asUppercase]
"Print it:
OrderedCollection ('FATHER' 'MOTHER' 'CHILD')"
```

produces an OrderedCollection with the values of the Dictionary object dic written in uppercase.

In the event that you want to loop through the associations rather than the values of a dictionary, you can use the message keysAndValuesDo :, which expects a block with two block parameters as an argument. For example, if we wanted to output the contents of our dictionary as a table in Transcript, we could proceed in the following way:

```
| dic |
dic := Dictionary new.
dic at: 'Kind' put: 'child'.
dic at: 'Mutter' put: 'mother'.
dic at: 'Vater' put: 'father'.
```

```
dic keysAndValuesDo:  
    [:german:english | Transcript cr; tab; show: german;  
     tab; show: english]
```

Note that the fact that `Dictionary` is a `Set` means that the order in which the associations are shown cannot be predicted.

It should be noted too that the message `keysAndValuesDo:` can be used with all ordered collections, such as arrays (see Sect. 10.2); in that case, the indexes assume the role of the keys.

10.2 Ordered Collections

The classes of ordered collections are subclasses of `SequenceableCollection` (see Fig. 10.2). The most prominent characteristic of these collections is that the components can be accessed using a numeric index. The class `Array` is a typical example. At this point, we'll again summarise their most important characteristics. In addition, we'll describe the classes `OrderedCollection` and `SortedCollection`. These three collection classes are heterogeneous, which means that they may contain any kind of objects as components.

`String`, `Symbol` and `Interval` are examples of homogeneous ordered collections; their components must always be members of the same class.

Common Messages for Ordered Collections

Table 10.3 provides an overview of a few important messages that can be used to access components in all ordered collections. To help you understand them better, Table 10.4 shows a few application examples for these messages. In this case, strings are used as sample receiver collections. These could be replaced though by instances of other subclasses of `SequenceableCollection`.

► **Note:** As we already mentioned in Sect. 8.1.8, since Version 7 of VisualWorks, all literals have become immutable objects. That means that all sample messages in Table 10.4 that modify their receiver cause an exception. You can avoid that result by first using a `copy` message to create a copy of the object you want to modify; the copy can be modified. We've omitted the `copy` messages from Table 10.4. The message `yourself` in the second example is necessary, because it is not the modified receiver that the message `at :put :` produces as a result, but rather its second argument.

Various messages are available for creating copies of collections (see Table 10.5). Table 10.6 shows application example.

Table 10.3 Messages for accessing components of ordered collections

Message Pattern	Meaning
at: anIndex	Yields the object of the collection with index anIndex
at: anIndex put: anObject	Writes the object anObject to the collection at position anIndex
atAllPut: anObject	Writes the object anObject to all components of the collection
first	Yields the first component of the collection; exception if the collection is empty
last	Yields the last component of the collection; exception if the collection is empty
indexOf: anObject	Determines the index of the object anObject within the collection; produces 0 if anObject is not present in the collection
replaceFrom: start to: end with: aCollection	Replaces the components of the collection with indexes start to end with the components of aCollection. The number of components of aCollection must be equal to end-start+1

Table 10.4 Application examples for the messages in Table 10.3

Expression	Result
'hallo' at: 2	\$a
'hallo' at: 2 put: \$e; yourself	'hello'
'hallo' first	\$h
'hallo' last	\$o
'hallo' indexOf: \$o	5
'123456789' replaceFrom: 3 to: 6 with: 'abcd'	'12abcd789'

Additional special messages also exist for enumerating elements. In addition to the do: message that can be used for all collection classes, you can also use the messages shown in Table 10.7 for enumeration. The following expression serves as an example for the use of the with:do: message:

```
#(2 4 6)
  with: #(3 5 7)
  do: [:i:j | Transcript cr; show: (i * j) printString]
```

The product of the corresponding array elements is output in Transcript. The result is shown in Fig. 10.10.

Table 10.5 Messages for copying ordered collections

Message Pattern	Meaning
copy	Creates an exact copy of the receiver collection
copyFrom: start to: end	Produces a new collection with the components of the index range start to end from the receiver collection
copyWith: anObject	Copies the receiver collection to a new collection to which the object anObject is added as the last element
copyWithout: anObject	Copies the receiver collection to a new collection, omitting every occurrence of the object anObject
, anOrderedCollection	Produces a new collection consisting of the concatenation of the receiver and the collection anOrderedCollection

Table 10.6 Application examples for the messages in Table 10.5

Expression	Result
#(1 2 3), #(4 5 6) copyFrom: 2 to: 4	#{2 3 4}
'hallo' copyWith: \$!	'hallo!'
'Five Chinese girls' copyWithout: \$i	'Fve Chnese grls'

Table 10.7 Messages for looping through ordered collections

Message Pattern	Meaning
reverseDo: aBlock	The elements of the receiver collection are enumerated in reverse order, beginning with the last element. For each element, the block aBlock (which must have a block parameter) is evaluated
keysAndValuesDo: aBlock	Like do:, loops through the receiver collection, but expects a block with two parameters; the index is assigned to the first and the associated element is assigned to the second
with: anOrderedCollection do: aBlock	Loops through in parallel the receiver collection and the parameter collection anOrderedCollection; it expects a block with two parameters, where the element of the receiver collection is assigned to the first and the corresponding element from anOrderedCollection is assigned to the second
findFirst: aBlock	Produces the index of the first element for which the evaluation of the block (1 parameter) produces true
findLast: aBlock	Produces the index of the last element for which the evaluation of the block (1 parameter) produces true

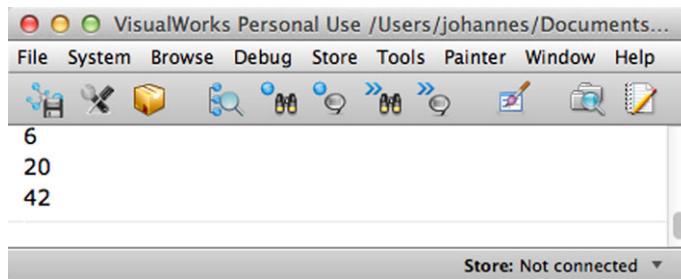


Fig. 10.10 Result of using a `with:do:` message

10.2.1 The Class Array

One special characteristic of this collection class in contrast to `OrderedCollection`s and `SortedCollection`s is that its instances have a fixed number of components. Once an array has been created, it can neither expand nor contract. This permits simple storage management, and so one uses arrays when the required size has been determined beforehand, and it is not necessary to have flexibility in varying the size. Unlike with sets, we cannot use messages like `add:` and `remove:` with arrays.

Arrays (along with `String` and `Symbol`) also differ from most other collection classes in that there are literal representations of array instances (see Sect. 3.2):

```
#('This is an array with' 3 'elements')
```

This is an option for creating instances of the class `Array`.

Creating Arrays

Although it is possible to use the expression `Array new` to create an array, it wouldn't make much sense to do so because the array would have no components. Using

```
Array new: 10
```

though creates an array with 10 components, whose value is `nil`. If one wants a different initial value for the array one could use the message

```
Array new: 10 withAll: 0.
```

In this case, all components have 0 as their initial value.

10.2.2 The Class `OrderedCollection`

During its existence, an instance of the class `OrderedCollection` can grow by having new elements added or shrink by have elements removed. Nevertheless, an `OrderedCollection` maintains the order in which the elements were added. The messages in Table 10.3 can be used for accessing components. In addition, you can use the messages `after:` and `before:,` which in each case expect an element of the receiver collection as an argument, and then respond with the element that occurs either immediately after or immediately before the argument. For example, the expression

```
#(2 4 6 8) asOrderedCollection after: 4
```

produces as a result the element 6, because the 6 occurs after the 4. An exception occurs if the `OrderedCollection` does not contain the object specified with `after:.` We've used a trick in this example to create an `OrderedCollection` object, because there are no literal representations for instances of this class: you can send the message `asOrderedCollection` to an array, which transforms it into an `OrderedCollection`. Section 10.3 describes other messages that can transform various types of collections into one another.

Adding Objects to `OrderedCollections`

We've already encountered the message `add:` in connection with unordered collections, which can also be used here. The elements transmitted as an argument is then always added at the end:

```
| oc |
c := #(2 4 6 8) asOrderedCollection.
c add: 10.
c      "Print it: OrderedCollection (2 4 6 8 10)"
```

Besides `add:,` you can also use the messages shown in Table 10.8 to add elements. Be aware, though, that these messages do not return the receiver collection as the result object but rather its argument. For example, evaluating

```
| oc |
oc := #(2 4 6 8) asOrderedCollection.
oc add: 10.      "Print it: 10"
```

Table 10.8 Messages for adding objects to ordered collections

Message Pattern	Meaning
add: obj1 after: obj2	Adds obj1 after the element obj2. An exception occurs if obj2 is not in the OrderedCollection
add: obj1 before: obj2	Adds obj1 before the element obj2. An exception occurs if obj2 is not in the OrderedCollection
addFirst: anObject	Writes the object anObject ahead of all existing components in the OrderedCollection
addLast: anObject	Writes the object anObject after all existing components in the OrderedCollection, same effect as add:
addAllFirst: anOrdrdCollection	Writes the elements of anOrdrdCollection at the start of the receiver collection
addAllLast: anOrdrdCollection	Writes the elements of anOrdrdCollection at the end of the receiver collection

produces 10 as the result and not OrderedCollection (2 4 6 8 10). It helps in some cases to append the message `Yourself`, which does nothing more than return to receiver as the result.

```
| oc |
oc := #(2 4 6 8) asOrderedCollection.
oc add: 10; yourself
"Print it: OrderedCollection (2 4 6 8 10)"
```

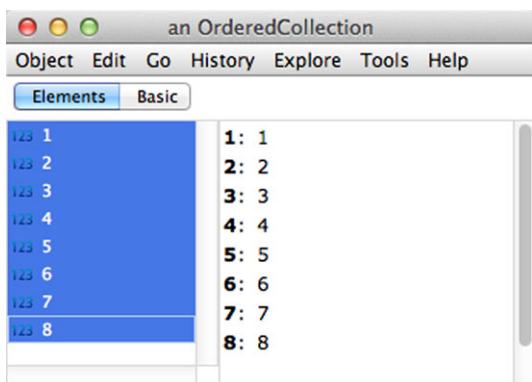
Every object understands the message `Yourself`.

The following sequence demonstrates a few of the messages listed in Table 10.8:

```
| oc |
oc := OrderedCollection new.
oc add: 5.
oc add: 8.
oc add: 6 before: 8.
oc addAllFirst: #(2 3 4).
oc addFirst: 1.
oc add: 7 after: 6.
oc      "Print it: OrderedCollection (1 2 3 4 5 6 7 8)"
```

Figure 10.11 shows this OrderedCollection in Inspector. The view in the **Basic** tab again gives insight into the internal management of an OrderedCollection object (see Fig. 10.12).

Fig. 10.11 An instance of the class `OrderedCollection`



Removing Elements from an `OrderedCollection`

We've already encountered the messages `remove:` and `remove:ifAbsent:` in connection with sets (see Sect. 10.1), and we can use them here in the same way. In addition, we can use the unary messages `removeFirst` and `removeLast` to remove the first and last elements.

Finally, we can use the message `removeAllSuchThat:` to remove all of those elements from an `OrderedCollection` for which evaluating the block argument yields `false`. The following sequence of messages shows an application:

```
| oc |
oc := OrderedCollection new.
```



Fig. 10.12 The internal structure of an instance of the class `OrderedCollection`

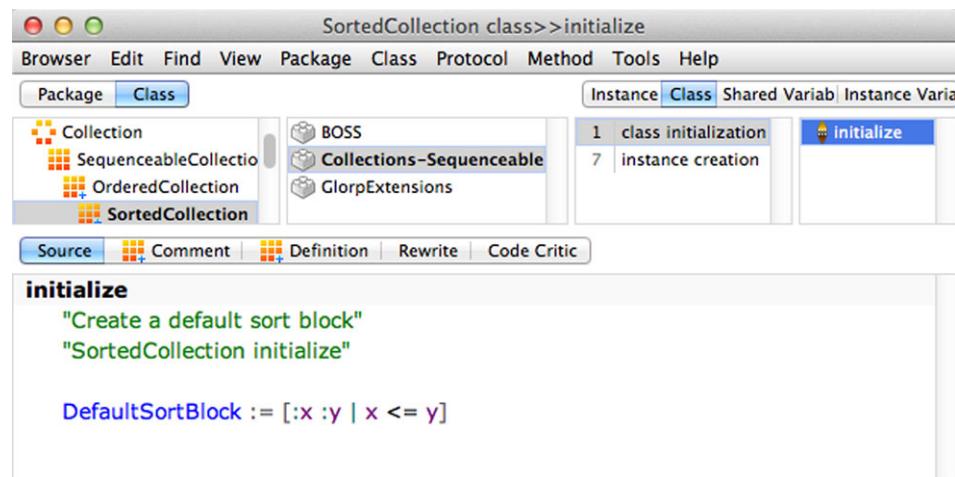


Fig. 10.13 DefaultSortBlock defines ascending sort order

```

oc addAllFirst: #(2 4 6 8).
oc removeAllSuchThat: [:elem | elem>5]
    "Print it: OrderedCollection (6 8)"

```

10.2.3 The Class `SortedCollection`

This refers to a subclass of `OrderedCollection` that contains collections whose elements are to be stored in a defined sort order. You can use these collections for all types of sorted lists.

The sort order is determined by a block that is stored in the instance variable `sortBlock`. Unless otherwise specified, this instance variable is populated with a so-called `DefaultSortBlock` when it is created. The class method `initialize` populates the shared variable `DefaultSortBlock` with this information (see Fig. 10.13), and specifies an ascending sort order. This is a block with two parameters, and the class `SortedCollection` ensures that the evaluation of this block always responds `true` for two successive elements of the collection. The user is responsible for seeing that the comparison operation used in the `sortBlock` is also defined for the elements of the collection.

In the following example, a `SortedCollection` is created with `DefaultSortBlock`, and afterwards whole numbers are added:

```
| sc |
sc := SortedCollection new.
sc add: 3; add: 1; add: 5.
sc      "Print it: SortedCollection (1 3 5)"
```

At any time though, you can use the set method `sortBlock:` to change the `sortBlock:`

```
| sc |
sc := SortedCollection new.
sc add: 3; add: 1; add: 5.
sc sortBlock: [:i:j | i >= j].
sc      "Print it: SortedCollection (5 3 1)"
```

Now, if we want to put instances of a class `Person` that has the instance variables `firstName` and `lastName` into a list sorted in ascending order according to last name, we could define the following `SortedCollection`:

```
| personList |
personList := SortedCollection new.
personList sortBlock: [:x:y | x lastName <= y lastName]
```

`sortBlock` indicates that the values of the instance variable `lastName` will be used for the comparison. Such a `sortBlock` definition basically makes the `OrderedCollection` a homogeneous data structure. All that remains is to add objects that understand the message `lastName`.

Now we can create `Person` objects and add them to the `SortedCollection` `personList`:

```
| personList |
personList := SortedCollection new.
personList sortBlock: [:x:y | x lastName <= y lastName].
p1 := Person new lastName: 'Luxemburg'; firstName: 'Rosa'.
p2 := Person new lastName: 'Liebknecht'; firstName:
'Karl'.
personList add: p1.
personList add: p2.
personList
```

Fig. 10.14 The Interval of whole numbers 1 to 10



When we use **Print it** to evaluate the sequence of messages, it displays the following OrderedCollection:

```
SortedCollection (Person with
Last name: Liebknecht
First name: Karl,
Person with
Last name: Luxemburg
First name: Rosa)
```

As an aside, we are assuming here that the class `Person` is able to use a suitable `printOn:` method (see Sect. 14.2).

10.2.4 The Class `Interval`

Instances of the class `Interval` are series of numbers that in their simplest form are created by specifying start and end values. To that extent, `Interval` objects are homogeneous collections. With arrays, they share the trait that—once created—new elements cannot be added to them nor can elements in them be removed.

Sections 4.2.2 and 8.1.5 have already discussed the multiple ways in which `Interval` objects can be used, especially for programming loops. Here we'll add just a few more aspects that we haven't examined yet.

The simplest way to create a series of numbers is to use the message `to:.` The receiver of the message `to:.` supplies the start value; the argument supplies the end value; the increment is 1. The expression

```
1 to: 10
```

defines the `Interval` object shown in Fig. 10.14. It represents the sequences of `SmallInteger` numbers from 1 to 10. As the figure shows, the number sequences are only ever stored by specifying the start value (`start`), end value (`stop`) and increment

(Step). You can use the `to:by:` message to establish the increment when you create an `Interval` object. The number sequence

```
1 to: 10 by: 3
```

contains the numbers 1, 4, 7 and 10.

The numbers used in the `to:` and `to:by:` messages may come from any `Number` class. The increment may also be negative. In that case, though, the end value must be less than the start value.

The expression

```
((1/3) to: 3 by: (2/3))
    do: [ :fraction | Transcript cr;
                      show: fraction printString]
```

writes the sequence of numbers

```
1
(5/3)
(7/3)
3
```

to Transcript.

The last example shows that one can, of course, also send the message `do:` to an interval. In Sects. 4.2.2 and 8.1.5, we used messages in the format

```
n to: m do: [ :i | <sequence of instructions> ]
```

We don't need to put parentheses around the `Interval` object simply because `Number` objects, as discussed in Sect. 8.1.5, also understand the messages `to:do:` and `to:by:do:.` We could have also written

```
((1/3) to: 3 by: (2/3))
    do: [ :fraction | Transcript cr;
                      show: fraction printString]
```

for the above example.

Otherwise, all messages for ordered collections can also be used for `Interval` objects.

10.2.5 The Class `String`

We've been using character-string literals in programming examples from the very beginning. The class `String` is a subclass of `Array` and `CharacterArray`. `String` objects share with arrays the fact that they can't expand and contract; they are, moreover, homogeneous collections, whose components are instances of the class `Character`.

At this point, we'll simply refer to the explanations in Sect. 8.1.8, where we have already discussed the most important messages that `String` objects can understand.

10.2.6 The Class `Symbol`

Symbols are character strings that can appear only once in the memory of a VM; strings, though, can have multiple identical copies. We've already discussed symbols in detail in Sect. 3.2. Behind the different ways that character strings and symbols are treated stands the more general principle of identity or equality of objects, which will be discussed more fully in Sect. 11.4.

Otherwise, symbols can be treated like strings.

10.3 Transforming Collections

On a few occasions, we've made use of the possibility of transforming one collection into a different kind of collection. Thus, for example, we can turn an `Interval` object into an instance of the class `OrderedCollection`:

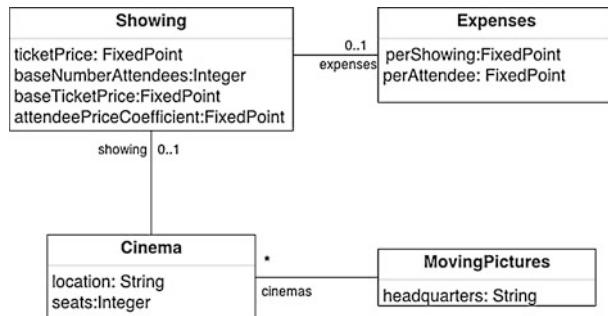
```
(1 to: 5) asOrderedCollection  
    "Print it: OrderedCollection (1 2 3 4 5)"
```

Strictly speaking, this does not actually transform the receiver collection; instead, a new collection of the requested type is created, which then contains the same elements as the original. In Sect. 11.4, we'll come back and consider the consequences.

You can use the following messages to transform almost any `Collection` object into the indicated object type:

- `asArray` into an array
- `asBag` into a Bag object
- `asSet` into a set

Fig. 10.15 Class diagram for the business *MovingPictures*



`asOrderedCollection` into an `OrderedCollection`
`assortedCollection` into an `SortedCollection`

You can supply the message `assortedCollection:` with a `sortBlock` block as an argument.

It is not possible to turn another collection into an `Interval` object.

You can use the messages `asString` and `asSymbol` to transform character strings into symbols and vice versa.

10.4 Case Study: Cinemas

This section picks up the case study from Sect. 7.2 and expands on it. In the meantime, the owner has expanded his business (*MovingPictures*) and now runs several cinemas in various cities. As a first step, we'll account for this by expanding the class diagram from Fig. 7.15 by an additional class, which represents the business itself and serves primarily to manage the various cinemas. For that, we'll need a new class **Cinema**. With these thoughts in mind, we've come up with the class diagram shown in Fig. 10.15.

The class **MovingPictures** has two instance variables:

`headquarters` This simply contains a character string with the location of the business's headquarters.

`cinemas` The various cinemas belonging to *MovingPictures* are stored in a collection. The instance variable is represented in the diagram by an association with the class **Cinema**. The asterisk on the association line means that an instance of *MovingPictures* can contain any number of **Cinema** instances.

The class **Cinema** has three instance variables:

`location` Character string with the name of the city where the cinema is located.

`seats` Number of seats in the cinema.

`showing` An instance of the class **Showing** will be created here, as it was developed in Sect. 7.2.

At this point, it's appropriate to say something about how classes are named. Basically, the name of the class `Showing` was derived from an analysis of the text that the cinema owner wrote defining his requirements for calculating the profit for a showing at a particular ticket price. In the context of the class diagram in Fig. 10.15 though, the name of the class is ambiguous. It would be logical to associate the idea of a *showing* with a movie showing. In fact, the instances of the class `Showing` serve the calculation of profits, and for that reason, the class should have a name that clearly indicates that fact. For that reason, we'll rename the class `ProfitCalculator`.

Renaming a class in this case—and in most cases—serves to improve the clarity of the program text. It's always worth undertaking such a task, because program texts are read far more often than they're written. Changes to a program that serves “merely” to improve its clarity or its structure without changing its functionality are referred to as *refactoring*. There are many reasons to undertake refactorings. Besides clarifying the name, they might also affect the class structure, the placement of methods within classes or the scope of methods. We won't go any deeper into this theme in this book, but interested readers should look at Fowler (2000).

Today it's considered good style in software engineering to perform refactorings whenever they appear necessary. Modern development environments should therefore also support the programmer in performing them. For the comparatively simple case of renaming a class, it's also important to check that this occurs consistently. In other words, it's not just the definition of the class that has to be changed, but also all places in the program where the class name is used, which must also be modified to suit the change.

The System Browser in VisualWorks permits the consistent renaming of a class. Once you have highlighted the class to be renamed, click on **Rename...** from the **Class** menu and key the new class name into the dialogue box that appears. The class definition will be changed, and the entire Image will be searched for places where the old class name was used so that the name can also be changed there. Figure 10.16 shows the definition of the class `ProfitCalculator` and Fig. 10.17 shows the correspondingly modified class diagram.

10.4.1 Assignment: Display all Cinemas in Transcript

The first programming assignment is to make it possible to output all cinemas with their locations and number of seats. It might look like this:

```
Hamilton 350
Mason 400
Clifton 250
Oakley 200
```

ProfitCalculator>accessing

Browser Edit Find View Package Class Protocol Tools Help

Package Class Instance Class Shared Variable Instance Variable

Cinemas (+50) Cinema 10 accessing
 Currencies CinemaNs 4 calculating
 Persons Expenses 1 initialize-release
 QuadEquats MovingPictures
 Glorp ProfitCalculator

Source Comment Definition Rewrite Code Critic

```
Smalltalk.CinemaNs defineClass: #ProfitCalculator
superclass: #{Core.Object}
indexedType: #none
private: false
instanceVariableNames: 'ticketPrice baseNumberAttendees baseTicketPrice
attendeePriceCoefficient expenses'
classInstanceVariableNames: ''
imports: ''
category: ''
```

Fig. 10.16 Class Showing renamed as ProfitCalculator

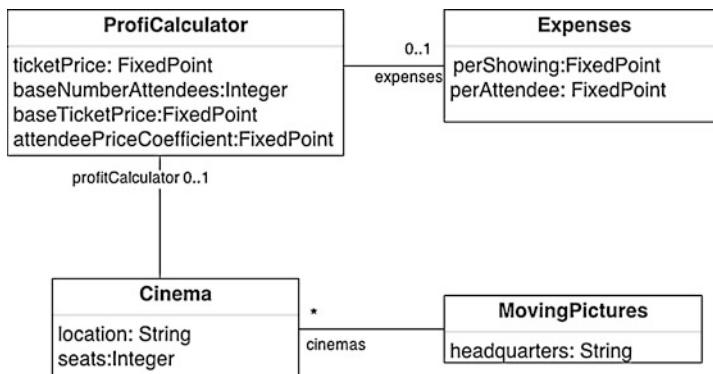


Fig. 10.17 Revised class diagram for the business *MovingPictures*

We'll use the technique that was introduced in Sect. 8.5.6 to write class methods as test methods. The class method² for the class *MovingPictures*

```
MovingPictures class>>showCinemas
"self showCinemas"
```

²You should again create this in a protocol called examples.

```
| bb |
bb := self new.
bb createExampleCinemas.
bb showAllCinemas
```

should therefore do the following:

1. Create an instance of the class `MovingPictures`.
2. By sending a message `createExampleCinemas`, cause the class instance to create the four cinemas shown above.
3. Finally, cause the output in Transcript.

To create the `MovingPictures` instance, we'll define a specific new method

```
MovingPictures class>>new
^super new initialize
```

with the boilerplate we're already familiar with. The `initialize` message sets sensible default values for the instance variables. The corresponding instance method might look like this:

```
MovingPictures>>initialize
self headquarters: ''.
self cinemas: OrderedCollection new
```

“Sensible default values” means that, after initialisation, the instance variables point to objects of the classes that were specified as attribute classes during the design (in the class diagram). Of course, instead of an empty character string for the variable `headquarters` we could enter the name of the city where the headquarters are located, as a sort of constant, which would only need to be changed if the business moved.

From the class diagram in Fig. 10.17, we can see that the purpose of the instance variable `cinemas` is to permit the assignment of any number of instances of `Cinema` to an instance of `MovingPictures`. But that is possible only if we implement the instance variable as a collection. In this case, we've selected the class `OrderedCollection`, which allows us to add or remove cinemas at any time.

► **Note:** The class `OrderedCollection`³ represents heterogeneous collections, that is, one whose elements can be instances of any kind of class. In our application, the el-

³See Sect. 10.2.2.

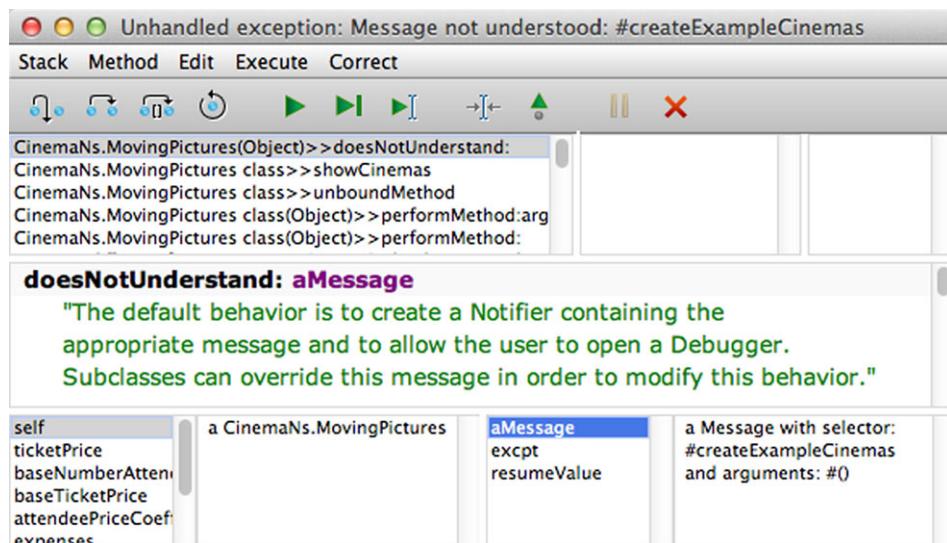


Fig. 10.18 The message `createSampleCinemas` was not understood

ements will, of course, only be instances of Cinema, at least according to the class diagram. It would be entirely conceivable though to define a special collection class as a subclass of `OrderedCollection` that might be called `CollectionOfCinemas`, which would ensure that the collection bound to the variable `cinemas` actually contained only instances of Cinema. We won't implement that for our little sample program though, because it would be of such limited use.

Defining Methods in the Debugger

The next method that we need is the instance method `createSampleCinemas`. Until we implement that method, whenever we use the method `showAllCinemas`, we will receive a message-not-understood exception (see Fig. 10.18). The debugger lets us create the “missing” method using **Define Method** under the **Correct** menu. Figure 10.19 shows the result. Since the debugger can, of course, not “know” what the method is supposed to do, the method body shows the expression `self halt`. If we activated the method in that state it would lead to the exception `Halt` encountered. But we'll replace the method body with the expression sequence

```
self
newCinemaIn: 'Hamilton' withSeats: 350;
newCinemaIn: 'Mason' withSeats: 400;
newCinemaIn: 'Clifton' withSeats: 250;
newCinemaIn: 'Oakley' withSeats: 200
```

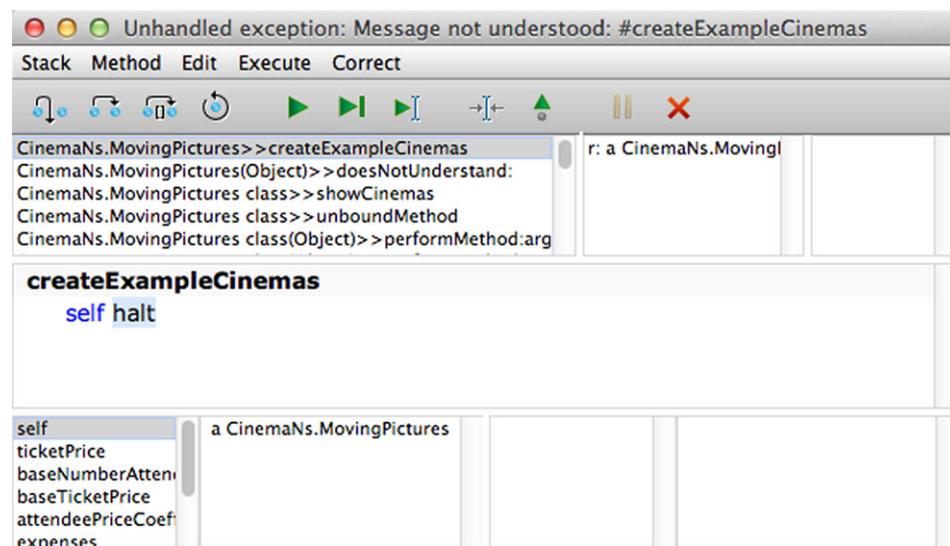


Fig. 10.19 The placeholder method `createSampleCinemas` was created

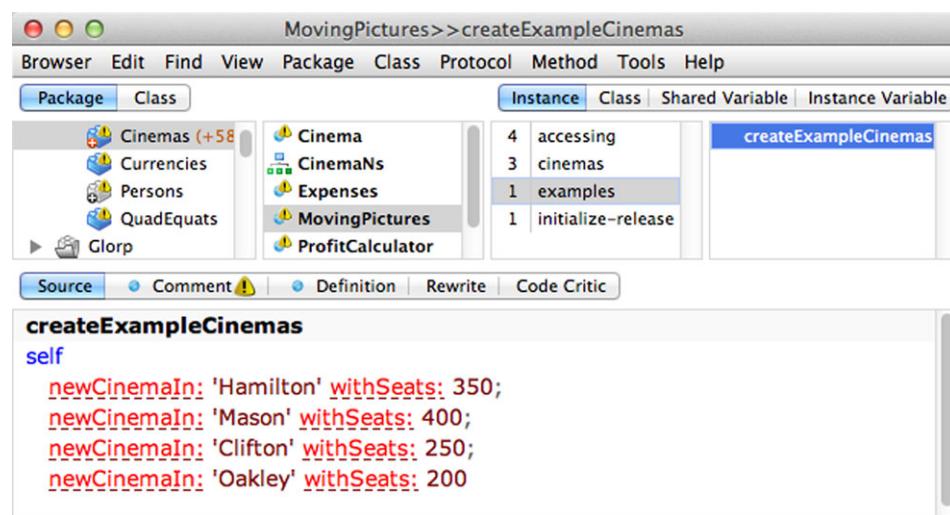


Fig. 10.20 Instance method `createSampleCinemas` has been defined

and click **Edit → Accept** from the menu. Then the debugger window of Fig. 10.20 is shown. At that point, we ignore the fact that `newCinemaIn:withSeats:` is a new message—one that we've only just “invented”. Now we've defined the `createSampleCinemas` instance method and the program can be continued. As always, the debugger highlights the next method to be sent, in this case, the message:

```
newCinemaIn: 'Hamilton' withSeats: 350
```

To proceed with the program, click **Execute → Run** from the menu. Since we still have no method for `newCinemaIn:withSeats:`, the program again stops with a message-not-understood exception. Once again, we can use the debugger to create the relevant method. We can continue in this way until all methods have been defined. In this way, the debugger also supports the top-down development of methods—a process that's very popular among Smalltalk programmers. At this point, we'll limit ourselves to supplying information on

Programming the Other Methods

as program text. We'll start with the method

```
MovingPictures>>newCinemaIn: aString  
           withSeats: anInteger  
self addCinema:  
    (Cinema in: aString seats: anInteger)
```

At this point, we'll introduce two new messages. It's very easy to program the method

```
MovingPictures>>addCinema: aCinema  
self cinemas add: aCinema
```

We use the message `add:` (see Sect. 10.2) to add an element to the `OrderedCollection` we already created in the instance variable `cinemas`, specifically, an instance of the class `Cinema`, which we previously created using the message `in:seats:`. The class method that belongs to this message

```
Cinema class>>in: aString seats: anInteger  
^self new location: aString; seats: anInteger
```

creates a new instance of `Cinema` and sets the instance variables using the corresponding set methods. It's also a good thing to define the `new` method for the class `Cinema`:

```
Cinema class>>new  
^super new initialize
```

And again the `initialize` method

```
Cinema>>initialize
self
location: '';
seats: 0;
profitCalculator:
    (ProfitCalculator withTicketPrice: 5.00s)
```

initialises the instance variables with meaningful initial values. In this case, the variable `profitCalculator` is supplied with an instance of the class `ProfitCalculator`, which corresponds to the class `Showing` that we discussed in Sect. 7.2.

In order for the method `MovingPictures class>>showCinemas` (see above) to run, we still need to implement the message `showAllCinemas`:

```
MovingPictures>>showAllCinemas
Transcript
cr;
show: 'MovingPictures has cinemas in: '.
self cinemas
do:
    [:cinema |
        Transcript cr; show: cinema showYourself]
```

First of all, a title is output to `Transcript`. Then, the `do:` message is sent to the `OrderedCollection` stored in the instance variable `cinemas`. The block sent as a parameter is thus evaluated for each cinema. The message `cinema showYourself` creates a string with the name and number of seats for each cinema; the message is then output to `Transcript` using `show::`.

The method

```
Cinema>>showYourself
^(self location copyWith: Character tab)
, self seats printString
```

accesses the name of the location and sends the message `copyWith: Character tab`⁴ to this character string; this creates a copy of the character string, to which a tab

⁴See Table 10.5.

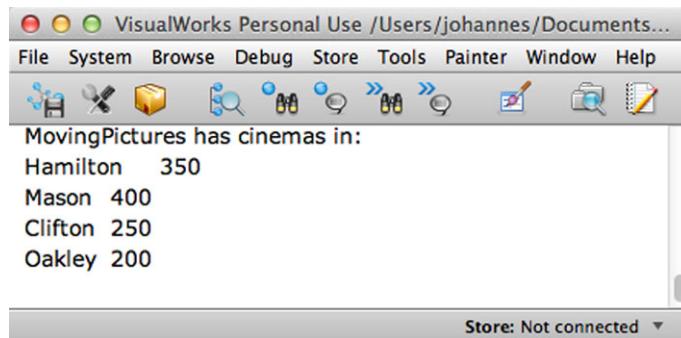


Fig. 10.21 List of cinemas shown in Transcript

character is appended. The message “,” appends the number of seats—transformed into a character string—to the whole string.

Now we've defined all of the methods necessary to run the method `MovingPictures class>>showCinemas`. The result should look like the screen in Fig. 10.21. Naturally, the method can run properly only when all of the required get and set methods have been properly created.

10.4.2 Assignment: Display the Profits of a Specific Cinema

We want the class `MovingPictures` to have a message available that can be used to determine the profit of a cinema in a specific location and with a specific ticket price. The message selector should look like this: `profitIn:atTicketPrice:.` If we continue to use the sample cinemas that we used in the last section, for each cinema at the same ticket price, we'll end up with the same value, because we supplied the same standard profit calculator (from Sect. 7.2) to each cinema. For example, if the ticket price is 5 €, the profit will be 414 €.

Once again we'll start with a class method from the class `MovingPictures` as a test program:

```
MovingPictures class>>showProfit
  "self showProfit"

  | bb profitInOakley |
  bb := self new.
  bb createExampleCinemas.
  profitInOakley := bb profitIn: 'Oakley'
                  atTicketPrice: 5.00s.
```

```

Transcript
cr;
show:
'Profit in Oakley at a ticket price of 5
Euros: ', profitInOakley printString

```

Once again we're using the sample cinemas from the previous section. The important (new) message in this case is `profitIn:atTicketPrice:`, which we must define as an instance method in `MovingPictures`.⁵

```

MovingPictures>>profitIn: aString atTicketPrice:
                        aFixedPoint
"calculates the profit of the cinema in aString at
ticketPrice aFixedPoint"

^(self cinemaIn: aString) profitAtTicketPrice:
                        aFixedPoint

```

We've introduced two new messages here:

`cinemaIn:` supplies the instance of the cinema in the location supplied as the argument for the message.
`profitAtTicketPrice:` The message is sent to the cinema determined via `cinemaIn:` and supplies the profit at the ticket price sent as the argument.

First let's look at the method

```

MovingPictures>>>cinemaIn: aString
"supplies the cinema in location aString"

^self cinemas detect:
[:cinema | cinema location = aString]

```

The purpose of the method is to determine the particular cinema from the `OrderedCollection` stored in `cinemas` for which `location = aString` is valid. We can, of course, use the `do:` message to program this:

⁵For this and subsequent methods, we'll practice good manners by explaining the purpose of a method in a brief comment.

```
MovingPictures>>cinemaIn: aString  
    "supplies the cinema in the location aString"  
  
    self cinemas do: [:cinema | cinema location = aString  
                           ifTrue:[^cinema]]
```

Once a cinema is found for which `location = aString`, `^cinema` causes us to leave the method with `cinema` as the response object. We're dealing here with the problem of fetching an element from a collection for which a specific condition is fulfilled; this is a typical application of the message `detect:.`⁶ For that reason, we use the first version of the method.

The Cinema object uses the method

```
Cinema>>profitAtTicketPrice: aFixedPoint  
    "calculates the profit when the ticket price  
     is aFixedPoint"  
  
    ^self profitCalculator profitAtTicketPrice:  
        aFixedPoint.
```

to delegate the calculation of the profit at a particular ticket price to its `ProfitCalculator` object. In other words, we must still define a method with the same name in the class `ProfitCalculator`:

```
ProfitCalculator>>profitAtTicketPrice: aFixedPoint  
    "calculates the profit when the ticket price  
     is aFixedPoint"  
  
    ^(self ticketPrice: aFixedPoint) profit
```

Using the method `profit` from the class `ProfitCalculator`⁷ to calculate the profit assumes that the ticket price is stored in the corresponding instance variable of the `ProfitCalculator` object. In this case, that's precisely what we do when we use the `set` method. One could object and say that the interface of the `ProfitCalculator` was clumsily chosen. It's entirely right to ask whether it makes sense to regard the ticket

⁶See Table 10.1.

⁷Or `Showing`, as the class was called in Sect. 7.2.

price as a component of the status of a `ProfitCalculator` object. It would be possible instead to always supply the ticket price as an argument, thereby disposing of the instance variable `ticketPrice`. Here, though, we won't undertake that change.

This finishes the implementation of the methods for our assignment. Using the method `showProfit` should now display this text in Transcript:

```
Profit in Oakley at a ticket price of 5 Euros: 414.00s
```

10.4.3 Assignment: Display all Cinemas Arranged by Size

Now we return to the task we set in Sect. 10.4.1, but will output to Transcript a list of cinemas arranged according to the number of seats. If we use the same cinemas, the list would look like this:

```
Mason 400  
Hamilton 350  
Clifton 250  
Oakley 200
```

The test method would look like this:

```
MovingPictures class>>showOrderedCinemas  
"self showOrderedCinemas"  
  
| bb |  
bb := self new.  
bb createSampleCinemas.  
bb showOrderedCinemas
```

All we have to do now is define the instance method `showOrderedCinemas`:

```
MovingPictures>>showOrderedCinemas  
Transcript  
cr;  
show: 'MovingPictures has the following cinemas',  
' arranged by number of seats: '.
```

```
(self cinemas assSortedCollection:  
           [:x:y | x seats >= y seats])  
do:  
    [:cinema |  
     Transcript  
        cr;  
        show: cinema showYourself]
```

This method differs from the `showAllCinemas` method defined in Sect. 10.4.1 only in the receiver of the `do:` message. Where we simply used `self cinemas` in `showAllCinemas`, in this case, we used the method `assSortedCollection:` to transform the `OrderedCollection` in `cinemas` into an `SortedCollection`.⁸ In this case, we cannot use the unary message `assSortedCollection:` though we can supply a sort block as an argument in which we have defined how to compare two `Cinema` instances. We select the comparison expression

```
x seats >= y seats
```

which sorts the `Cinema` objects in descending order.

Then, as always, we send the `do:` message to the `SortedCollection` we just created so that they can be output. The result in `Transcript` now looks like this:

```
MovingPictures has the following cinemas arranged by  
number of seats:  
Mason 400  
Hamilton 350  
Clifton 250  
Oakley 200
```

⁸See Sect. 10.2.3.

This chapter picks up on various aspects of Smalltalk and the class hierarchy. These topics don't have any direct thematic connection, but they all share the fact that they've been touched on in previous chapters. This chapter looks at them in greater detail. In the final section on object identity, we do in fact look in detail at a new topic that we haven't considered before.

For example, blocks were discussed as part of case-by-case distinctions (see Sect. 3.1.2) and repetitions (see Chap. 4). Section 11.1 treats their much greater significance.

Section 11.2 explains more deeply and also in context how to find the method that suits a message, especially in cases where the method directory for the class of the receiver object contains no method with the name of the message selector.

The next section (Sect. 11.3) examines the question: Exactly what do classes look like whose instances are themselves classes? This question arises because of the basic Smalltalk rule that classes are also objects.

Finally, Sect. 11.4 investigates the difference between sameness and equality with respect to objects, and also how to create copies of objects.

11.1 Blocks

11.1.1 Blocks as Objects

Syntactically, a block is a sequence of messages enclosed between square brackets; interestingly, the evaluation of the block produces the block itself. Where the evaluation of the expression

```
3 * 5
```

of course produces an instance of the class `SmallInteger`, the evaluation of the expression

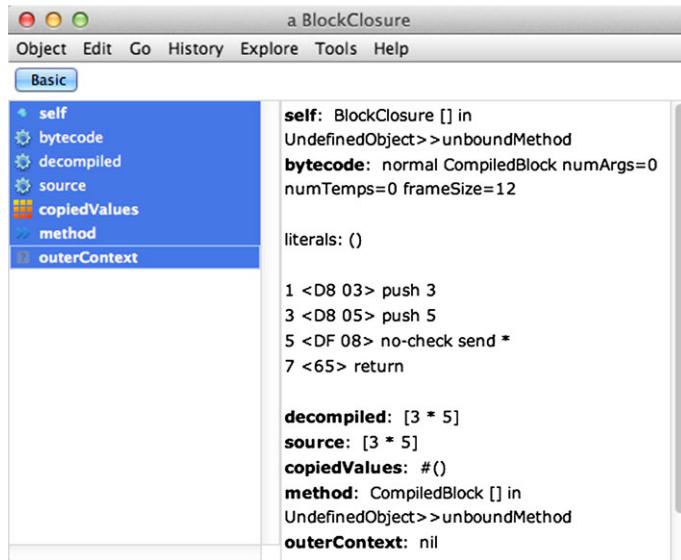


Fig. 11.1 A block is an instance of the class `BlockClosure`

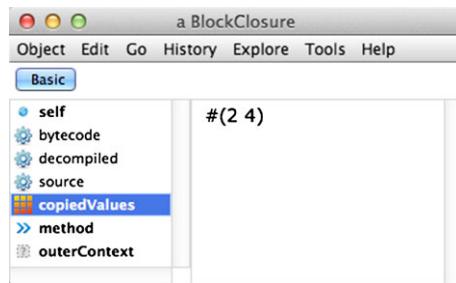
```
[3 * 5]
```

leads to an instance of the class `BlockClosure`, which the Inspector will convince you of (Fig. 11.1). The representation in the **Basic** view of the Inspector shows the aspects `bytecode`, `decompiled` and `source` for `BlockClosure` objects. Just as is true for `self`, which is shown for all objects, these are not instance variables. Under the aspect `source`, we find the source text in which the block is embedded, in this case, the method `unboundMethod` that is assigned to the Workspace. Under `bytecode`, one finds the `bytecode` that the compiler created as it translated the expressions within the block.

A `BlockClosure` object possesses the instance variables `method`, `outerContext` and `copiedValues`. In the final analysis, these contain the information that the VM needs to evaluate the content of the block when a `value` message is sent to it. The values of the instance variables for the block `[3 * 5]` can be found in Fig. 11.1.

The variable `method` contains an instance of the class `CompiledBlock`, which ultimately is the result of the compiler's translation of the block. Without going into too many further details, we'll just say this: Translated message sequences (blocks or methods) are—like everything else in Smalltalk—objects. Furthermore, blocks are instances of the class `CompiledBlock`, and methods are instances of the class `CompiledMethod`. Putting it somewhat simply, the other two instance variables contain information about the textual context of the block, which will be necessary if the block must be evaluated when it receives a `value` message. In our example though, no additional information is

Fig. 11.2 A block with copied values



necessary, because the result of the evaluation of the expression `3 * 5` is independent of the context in which the evaluation occurs.

Without going into detail, let's look at an otherwise useless example of code that contains a block in which the variable `copiedValues` is populated with a value:

```
| a b block |
a := 2.
b := 4.
block := [a + b]
```

If we look at the variable `block` in Inspector (see Fig. 11.2), we see that an array containing the values of `a` and `b` at the moment the block was translated was placed in the variable `copiedValues`.

When we speak of the *outer context*, we mean the method or the block (since blocks can also be nested) in which the block appears. The context information that a block carries with it also includes, among other things, the instance variables and local variables of a method of the object that contains the block. This context information is stored in the variable `outerContext`. The context information also permits a return (^) from the block to occur. If a block containing a return symbol is evaluated within a method, the method in which the block was defined is left.

11.1.2 Blocks with Parameters

Blocks share some features with methods:

- They bracket a series of messages that can be activated at any time. Methods are activated by sending a message with the same name; blocks are activated by sending one of the variants of the `value` message (see below).
- Like methods, blocks can have parameters for which arguments must be supplied at activation.
- Temporary variables can be defined within blocks.

- Evaluation also yields a result object. It is the object that results from the evaluation of the last expression within the block. If a return symbol is used, it must occur before the last expression.

We already saw blocks with parameters when we discussed repetitions in Chap. 4. In the expression

```
#(3 4 5) do: [:elem | Transcript show: elem printString]
```

the `do:` message expects a block with exactly one parameter as an argument. The elements of the receiver collection are then assigned in turn to this block parameter.

The block parameters are always declared between the opening square bracket and a pipe. In the declaration (and only there), each identifier is preceded by a colon. The scope of block parameters is limited to the surrounding block.

In the following simple example, the message `value:` is sent to a block with a single parameter:

```
| a |
a := 1.
[:increment | a := a + increment] value: 3.
Transcript show: a printString.
```

When the block is evaluated, the argument transmitted with the message replaces the block parameter, so that in the example the number 4 will be output in Transcript.

Blocks may also contain more than a single parameter. The following Smalltalk expression

```
[ :x :y | x@y] value: 100 value: 50
```

creates a `Point` object with the coordinates 100 and 50.

Table 11.1 shows variants of the `value` message for different numbers of block parameters.

If temporary variables with a scope limited to the block are to be used within a block, they are declared in the customary fashion between two pipes; if they are used, they occur after the block parameters. In the following example, a temporary variable named `aPoint` is declared:

Table 11.1 Messages for sending parameters to blocks

Message Pattern	Meaning
value: anObj	For an object to be sent
value: obj1 value: obj2	For two objects to be sent
value obj1 value: obj2 value: obj3	For three objects to be sent
valueWithArguments: argArray	For more than three (but at most 255) objects to be sent

```
[ :x :y | | aPoint |
aPoint := x@y.
aPoint translatedBy: 200@0] value: 100 value: 50
```

Evaluating this expression produces the `Point` object `300@50`.

11.1.3 Applications

The basic purpose of blocks is to allow programmers to define series of messages whose evaluation is to be delayed. There are many possibilities for applying this principle.

Case-by-Case Distinctions

An important use is as arguments for messages that are used to make case-by-case distinctions (`ifTrue:`, `ifFalse:`, etc. See Sect. 3.1.2). In the expression

```
(a ~= 0) ifTrue: [b := 1.0/a]
```

the decision whether the block should be evaluated can only be reasonably determined within the `ifTrue:` method.

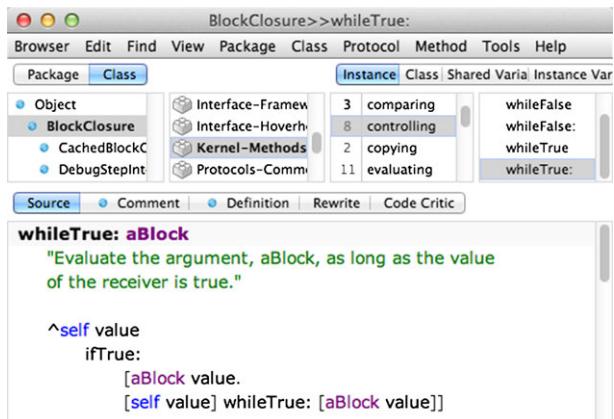
Normally, the argument expressions in a message are evaluated before sending the message. In other words, the method receives the *evaluated* arguments. This is a technique called *strict* or *eager evaluation*. The square brackets delay the evaluation, so that the method receives the *unevaluated* arguments. This technique is called *delayed* or *lazy evaluation*.

Lazy evaluation is much more significant than its use in implementing case-by-case distinctions in Smalltalk. For example, it's an important technique in functional programming. Section 12.4 contains a small example of its use in this application.

Repetitions

Sending an unevaluated argument to a method does more than simply allowing the method to determine if the argument should be evaluated once or not at all, as occurs with

Fig. 11.3 The method `whileTrue:` in `BlockClosure`



`ifTrue::`. It also allows the argument to be evaluated more than once. Repetition messages are based on this technique, and in this instance, we'll make use of a technique that up to now we haven't seen. As the following example shows, within a block, it's also possible to send the block itself a `value` message.

```

| block i |
i := 0.
block := [i := i+1.
Transcript show: i printString; cr.
block value]

```

If we were to use the message `block value` to evaluate the block stored in the variable `block`, we would end up with an endless output to the Transcript, because the block would continue to evaluate itself after its initial evaluation. We could only stop this endless loop by sending a User Interrupt (Ctrl + Y). The principle of *recursion* that's illustrated here (a block calls upon itself, see Chap. 12) is at the basis of all repetition messages in Smalltalk. Figure 11.3 shows as an example the method `whileTrue:` in the class `BlockClosure`.

The receiver of a `whileTrue:` message must always be a block whose evaluation produces `true` or `false`. You can see in the method body that the expression `self value` first evaluates the receiver block. If the result is `true`, the parameter block `aBlock`, which contains the message sequence that's to be repeated, is evaluated once. After that, the `whileTrue:` message is called recursively. The process ends when the expression in the first line (`self value`) produces `false`.

Individual Object Behaviour

The behaviour of objects is determined by their reaction to messages, which has been established in the methods of the class in common for all objects of the class. This is the way

it is supposed to work for most applications. But it can also happen that a situation arises in which you might want to give individual objects of a class a type of behaviour that's different from that of other instances of the same class. Looking at in a purely technical way, this could be realised in the following way:

1. Define one or more instance variables that can take blocks.
2. Define set methods as you normally would for these instance variables.
3. Finally, construct a method where evaluating the blocks permits the performance of the behaviour stored in the instance variables.

This will be illustrated in the next section using a small application example.

11.1.4 Case Study of a Finite-State Automaton¹

A finite-state automaton is a mathematical concept of great importance in computer science in which there are many practical applications, such as in hardware design or in the syntax analysis of programming texts. Automata theory uses several variants of finite-state automata. For us, it's enough to provide a Smalltalk implementation for one variation based on a descriptive explanation.

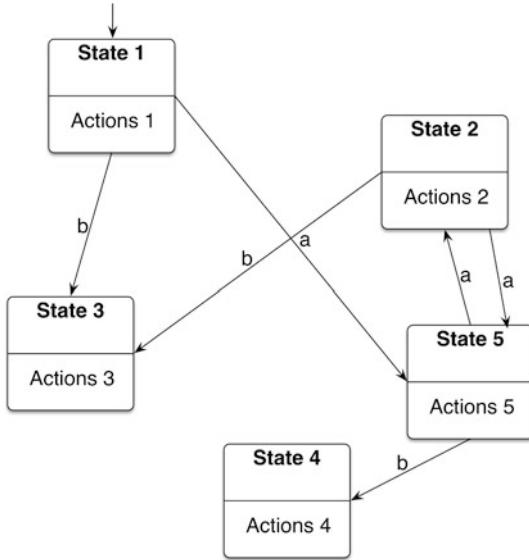
A finite-state automaton is an abstract machine that consists of a set of states. At any one time, the automaton is in precisely one state. A series of actions is connected with each state, which are executed when the automaton enters that state. Further, in each state, the automaton processes a so-called input symbol from a finite series of such symbols. The input symbol determines the next state, into which the automaton enters after having finished processing the series of actions. The sequence of states always begins its run in a defined starting state and ends after the automaton has completely processed the series of input symbols.

Finite-state automata are often portrayed using so-called state-transition diagrams (or briefly, state diagrams). Figure 11.4 shows such a diagram. States are shown as rounded squares, state transitions by arrows and the starting state by an entering arrow (State 1). The input symbol that causes one state to transition to another is written beside the applicable arrow. At this point, and in the following text, we assume, for simplicity's sake, that the input symbols are individual characters or character strings.

Note that an automaton is completely defined when a subsequent state has been defined for each input symbol and each state. This is not true in the case of the diagram in Fig. 11.4. If, for example, the automaton is in state 4, no subsequent state has been defined if the sequence of input symbols has not been exhausted.

A simple application example for a finite-state automaton is a storage element that's commonly used in digital technology called the JK flipflop. The JK flipflop stores one

¹The information in this section is based on Ivan Tomek's *The Joy of Smalltalk* (<http://live.except.de/doc/books/JoyOfST/JoyOfST.pdf>).

Fig. 11.4 A state diagram

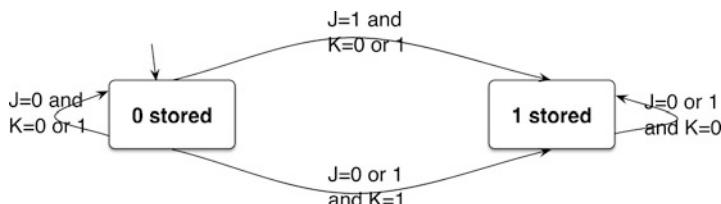
bit; that is, it can be in one of exactly two states. The state transitions are controlled by two entry points, called J and K, which can have the values 0 or 1. Figure 11.5 shows the conditions for the state transitions. The state symbols contain no actions, because the flipflop does nothing but change its state.

A finite-state automaton is defined by the following aspects:

- The set of states together with the actions associated with them
- The state transitions resulting from processing the input symbols
- The starting state

If we examine a particular point in time in the automaton's running process, additional aspects are added, such as the *current state* in which the automaton finds itself at the moment of examination, and the series of input symbols, which the automaton is intended to process.

Based on these considerations, we have the following class definition for a finite automaton:

**Fig. 11.5** Switching behaviour of the JK flipflop as a finite-state automaton

```
Smalltalk.AutomataNs defineClass: #Automaton
    superclass: #{Core.Object}
    inexactType: #none
    private: false
    instanceVariableNames: 'stateSet startState
    currentState '
    classInstanceVariableNames: ''
    imports: ''
    category: ''
```

Before we start to define methods for the class, let's look a little more closely at the essence of states. Based on our thoughts up to this point, a state is characterised by three aspects:

- An identifier (e.g., “State 1” or “0 stored”)
- The series of actions
- A kind of table that for each input symbol specifies what the next state is

A state is thus itself a complex construct, so that it seems practical to view states as instances of the following class:

```
Smalltalk.AutomataNs defineClass: #State
    superclass: #{Core.Object}
    inexactType: #none
    private: false
    instanceVariableNames: 'name actions nextStates '
    classInstanceVariableNames: ''
    imports: ''
    category: ''
```

In the instance variable `name`, we store a string for the identifier for the state. In `actions`, we must store a Smalltalk sequence of messages in the form of a block representing a series of actions. This will make it possible to provide an individualized behaviour for each instance of this class. This is necessary, because each state has to execute its own series of actions once it becomes the current state.

We have to put the table we mentioned earlier in the instance variable `nextStates`. This table will contain associations in the format

$$\text{Input symbol} \rightarrow \text{Next State}$$

For storage of any number of such associations—we also call them *key-value pairs*—the Smalltalk class library has a standard solution: the class `Dictionary` (see Sect. 10.1.3).

Using the expression

```
aDictionary at: $x put: aState
```

we store a new assigned pair with the key \$x and the value aState in the dictionary aDictionary. The expression

```
aDictionary at: $x
```

provides the value (the state) that is stored in the Dictionary under the key \$x.

Methods of the Class Automaton

Now let's consider the methods of our class Automaton. First of all, we'll create a class method that will allow us to create and initialise a new automaton:

```
withStateSet: arrayWithStates startState: aState
    "creates a new automaton with the state set
     arrayWithStates and the start state aState"
^self new
    stateSet: arrayWithStates
    startState: aState
```

That method uses, in turn, this instance method:

```
stateSet: arrayWithStates startState: aState
    "initializes an automaton (receiver)"

self
    stateSet: arrayWithStates
    startState: aState
```

At this point, the instance variable currentState remains uninitialised; the initialisation occurs when the automaton is started (see below). We will not bother to describe the get and set methods for the instance variables.

Next we'll examine the method that supplies the automaton with a series of input symbols to be processed and then starts the processing:

```
processInput: arrayWithEntrySymbols
    "starts running the automaton for the series of
     input symbols in arrayWithEntrySymbols"

    self currentState: self startState.
    self currentState process.
    arrayWithEntrySymbols do:
        [:currentEntrySymbol |
         self nextState: currentEntrySymbol.
         self currentState process]
```

The following simple algorithm lies at the base of this method. First off, the current state of the automaton is initialised with its starting state, and its series of actions is started by sending the message `process`. After that, the following steps must be repeated for each input symbol:

1. The next state for the current input symbol must be determined. To do that, the automaton sends itself the message `nextState:`, using the current input symbol as an argument.
2. The new current state must now be prompted to execute its actions. For that reason, it is again sent the message `process`.

The automaton's run ends automatically when the series of input symbols is exhausted.

The method `nextState` looks like this:

```
nextState: anObject
    "determines the next state from the current state and
     the input symbol anObject."

    self currentState:
        (self currentState nextState: anObject)
```

In this case, the determination of the next state is delegated by sending a message of the same name to the current state object.

Methods of the Class **State**

Now we come to the methods of the class `State`, where first of all we consider the class method used for creating a new state:

```

new: aString
    "creates a new state with the name aString"

    ^self new initialize: aString

```

The `initialize:` message sent in that method is based on the instance method:

```

initialize: aString
    "initializes State (receiver) with the name aString
     and an empty Dictionary for the next states"

    self name: aString.
    self nextStates: Dictionary new

```

Finally, that method sets the name and initialises the instance variable `nextStates` with an empty instance of the class `Dictionary`.

We won't discuss the get and set methods for this class.

The instance method `nextState:` uses the input symbol sent as an argument to access the dictionary `nextStates` and thus determines the next state:

```

nextState: anObject
    "determines the next state for the input symbol
     anObject."

    ^self nextStates at: anObject

```

To fill the dictionary `nextStates`, the following method is used:

```

nextState: aState for: anInput
    "defines the next state of the receiver for an input"

    self nextStates at: anInput put: aState

```

The last method that we need is `process`, which causes a state to process its own series of actions. And it too is as simple as can be:

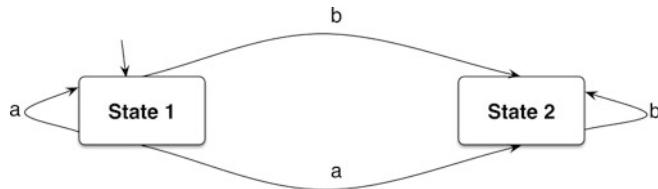


Fig. 11.6 A simple automaton

```

process
  "executes the messages stored
  in the instance variable actions"

  self actions value
  
```

Because we assume that the series of actions is stored in the instance variable `actions` in the form of a block of Smalltalk messages, we need only to send the message `value` to this block.

Sample Automata

Now we have assembled all of the methods needed to build an automaton and process a series of symbols. First of all, we examine a simple automaton consisting of two states that processes the input symbols \$a and \$b and has the state diagram shown in Fig. 11.6. The actions in the states will each only create an output in Transcript that will then show which states the automaton passed through.

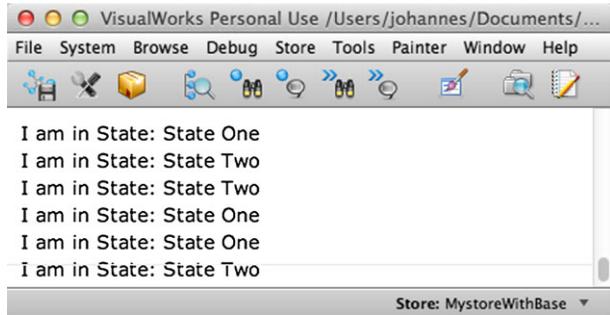
To build this automaton, we'll create the following class method for the class `Automaton`:

```

simpleAutomaton
  "Automaton simpleAutomaton"

  | z1 z2 automaton |
  z1 := State new: 'State One'.
  z2 := State new: 'State Two'.
  z1 nextState: z1 for: $a.
  z1 nextState: z2 for: $b.
  z2 nextState: z2 for: $b.
  z2 nextState: z1 for: $a.
  z1 actions:
    [Transcript
      cr; show: 'I am in State: ', z1 name].
  
```

Fig. 11.7 Result of running the simple automaton



```

z2 actions:
[Transcript
    cr; show: 'I am in State: ', z2 name].
automaton := Automaton
    withStateSet: (Array with:z1 with:z2)
    startState: z1.
automaton processInput: #($b $b $a $a $b)

```

The first thing that occurs here is the creation of two states and the definition of the next state for each. The action of each state consists of writing its name to Transcript. At the end the automaton is created and a `processInput:` message is sent to it with the series of symbols

```
b b a a b
```

If we execute this method by evaluating the expression

```
Automaton simpleAutomaton
```

The output shown in Fig. 11.7 appears in Transcript.

We can test the JK flipflop (see Fig. 11.5) with the following class method:

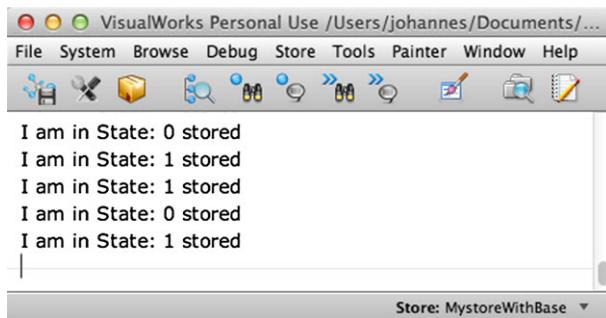
```

jkFlipFlop
"Automaton jkFlipFlop"

| z1 z2 automaton |
z1 := State new: '0 stored'.

```

Fig. 11.8 Sample run of the JK flipflop



```

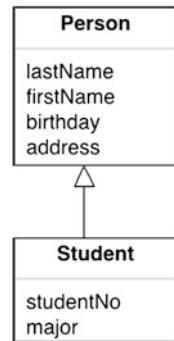
z2 := State new: '1 stored'.
z1 nextState: z1 for: '00'.
z1 nextState: z1 for: '01'.
z1 nextState: z2 for: '10'.
z1 nextState: z2 for: '11'.
z2 nextState: z2 for: '00'.
z2 nextState: z2 for: '10'.
z2 nextState: z1 for: '01'.
z2 nextState: z2 for: '11'.
z1 actions:
    [Transcript
        cr; show: 'I am in State: ', z1 name].
z2 actions:
    [Transcript
        cr; show: 'I am in State: ', z2 name].
automaton := Automaton
    withStateSet: (Array with:z1 with:z2)
        startState: z1.
automaton processInput: #('10' '11' '01' '11')

```

The result is shown in Fig. 11.8. The input symbols in this case consist of character strings containing two characters each—0 or 1—that each represent the signal at the J or K entry to the JK flipflop.

Even if the “individuality” of the behaviour of the instances of the class `State` is not very apparent, we can nevertheless see that the blocks stored in the instance variable `actions` can contain any series of message so that each state can react differently when it receives the message `process`.

Fig. 11.9 Two classes for people and students



11.2 Inheritance—Method Search

Among the most difficult aspects of object-oriented programming to understand is the way to determine a method to be activated that belongs to a message. We call this process *method search*.

Of course, there are trivial examples of method searches. When a message is sent to an object, a search is made in the list of methods for the class of the receiver object to see whether a method exists with the name of the message selector. If that is the case, the method is activated and the method search ends.

It gets more complicated though when the method can't be found in the directory of the receiver class. As we know, a class inherits the methods of its superclass, and so the search needs to continue in the directories of the superclasses.

We've already discussed some aspects of this topic in Sects. 8.3 and 8.4.

At this point, we'll examine the problem in greater detail using a simple class hierarchy that's shown in Fig. 11.9. We can see that people are described in general terms based on characteristics like first and last name, birthday and address. For students, we also include the student number and major field of study.

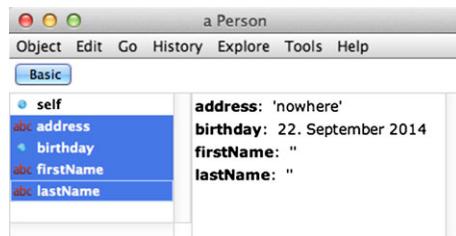
First we'll create a class method new for the class Person, which creates a new Person object and populates the instance variables with default values:

```

Person class>>new
    "creates a new person and initializes the
     instance variables"
    ^super new initialize
  
```

The expression `super new` causes the creation of an uninitialised object for the class Person. Using the pseudo-variable `super` rather than `self` causes the search for the method `new` to start in the superclass of Person, that is, in `Object`. That's necessary

Fig. 11.10 A new Person object



in this case too; otherwise, the new method would be found in the method directory for `Person`, and would thus continually invoke itself.

The instance method `initialize` initialises the instance variables for a `Person` object, which will not be discussed here in greater detail:

```
Person>>initialize
  self lastName: ''.
  self firstName: ''.
  self birthday: Date today.
  self address: 'nowhere'
```

And so if we execute the expression

```
Person new
```

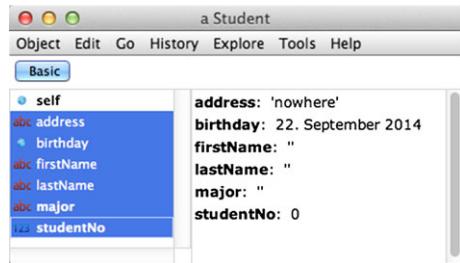
an instance of `Person` is created, as shown in Fig. 11.10.

Now let's look at the creation and initialisation of an instance of the class `Student`. The inherited instance variables of a `Student` object are to be initialised in the same way that it happened for persons. In addition—that's the hypothetical requirement at least—the student number will be set to 0 and the major to an empty string. This can be accomplished using the following method:

```
Student>>initialize
  super initialize.
  self studentNo: 0.
  self major: ''
```

Note that the receiver of the message that activates this method is an instance of the class `Student`. That is, the two pseudo-variables `self` and `super` both point to this `Student` object (see Sect. 11.2.2). The expression

Fig. 11.11 A new Student object



```
super initialize
```

causes the message `initialize` to be sent to this object, but the search for the correct method—because of the use of `super` instead of `self`—begins in the superclass. That is, the `initialize` method defined in the class `Person` is executed for the `Student` object, which causes the initialisation of the inherited instance variables. Finally, the set methods for the variables `studentNo` and `major` are activated.

The creation of a new, initialised `Student` instance will now be achieved by defining an appropriate `new` method:

```
new
^super new
```

► **Note:** This kind of a `new` method is actually superfluous, because all it does is activate the method with the same name in the superclass, which would happen even if this method didn't exist in the class `Student`. In this case, it simply helps us better illustrate method searching, which an example in the next section will run through step by step.

Using `super` here activates the `new` method in the class `Person`. Since `super` is bound to the class `Student`, it first creates an uninitialised `Student` instance, to which the message `initialize` is sent. Since the receiver of this message is a `Student` object, the `initialize` method is searched for in the class `Student`, where it is also found.

And so, if you execute the expression

```
Student new
```

a `Student` instance like the one in Fig. 11.11 is created.

11.2.1 Rules for the Method Search

At this point, we'll review the rules that are used when an object receives a message and has to determine which method to activate.

Let's start with the expression

```
object message
```

Assume that `ObjectClass` is the class of the object `object`. In that case, the method to be activated for the message selector `message` is determined by the following procedure:

1. Does the method protocol for `ObjectClass` contain a method called `message`?
2. If yes, the method being looked for has been found and is activated.
3. If no, check to see if `ObjectClass` is identical to the class `Object`.
 - a. If it is, then no suitable method exists in the class `Object` either. In other words, the object called `object` does not understand the message called `message`, and the exception “Message not understood” is displayed.
 - b. If it is not, replace `ObjectClass` with its superclass and return to Step 1.

11.2.2 The Meaning of the Pseudo-variables `self` and `super`

At this point, we'll review systematically the way that the use of the pseudo-variables `self` and `super` affects how the method search proceeds.

First of all, it's true that both pseudo-variables are always bound to the same object, which is the receiver of the message for which the method was activated in which `self` or `super` was used.

As an illustration, let's look at the following example. We'll use the classes `Person` and `Student` that we introduced at the beginning of Sect. 11.2.

```
| p |
p := Person new.
p firstName: 'Rosa'
```

At the moment that the method `firstName:` in the class `Person` is activated, the memory status looks like Fig. 11.12. The `Person` object bound to variable `p` is now available via the two pseudo-variables.

The only difference between `self` and `super` lies in how the method search proceeds when one uses one of the two pseudo-variables in the receiver position in a message expression:

Fig. 11.12 self and super always point to the same object

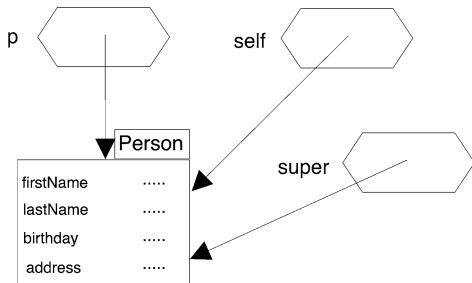
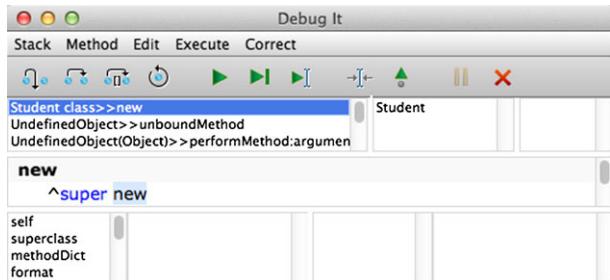


Fig. 11.13 Activating the class method new in the class Student



`self` in the receiver position causes the method search for the message to begin in the class of the object to which `self` is bound.

`super` in the receiver position causes the method search for the message to begin in the superclass of the class that contains the method in which `super` is used.

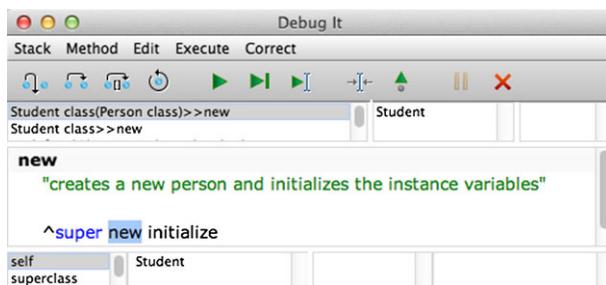
The rule for `super` sounds a little complicated. The superclass of the class that contains the method in which `super` is used is in many—but not all—cases the same as the superclass of the class of the object to which `super` is bound. Once again, we'll provide an example to show the difference.

If we use **Debug it** to execute the expression

```
Student new
```

and click **Step Into** once the debugger has opened, Fig. 11.13 appears. We can see here that `self`—and therefore `super` too—is bound to the class `Student`, because `new` is a class method. Based on the rule stated above, the method search starts in the superclass of the class that contains the method in which `super` is used. In this case, `super` is used in a class method of the class `Student`, and so the search begins in the class methods of the class `Person`. In this case, that class is also the superclass of the object to which `super` is bound.

Fig. 11.14 Activating the class method new in the class Person



And now if we activate the class method new (using **Step Into**), we see in Fig. 11.14 that `self` and `super` still point to the class `Student`. In the first line of the method stack, we also see that the class method new of the class Person is in the process of being executed with the class `Student` as receiver. At this point, the expression

```
super new
```

again appears, and we can also see clearly why the rule for a method search for `super` appears to be so complicated. The method in which `super` is used here is a class method of the class Person, and so the method search starts in the superclass of Person, which is the class Object. This is, of course, also the method we are searching for. But the class Object is not the superclass of the class `Student`, the class of the object bound to `super`. `Student`'s superclass is Person, and if we were to use Person's class method new, we would find ourselves in an endless recursion.

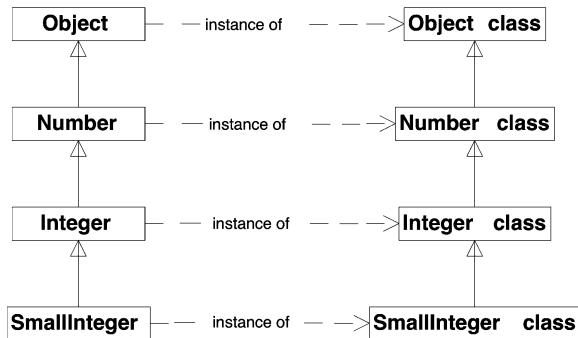
11.3 Metaclasses

Unlike other popular object-oriented programming languages like C++ or Java, in Smalltalk, classes are also objects. That means that classes are first-class runtime objects. Classes can send and receive messages and are thus, like other objects, able to be objects of calculation operations even at runtime. The main task of classes at a program's runtime is to create instances of themselves. In C++ and Java, in which classes no longer exist at runtime, instance creation is integrated into the programming language itself. Whenever a class is declared, so-called *constructors* automatically become available.

If classes are objects and, conversely, each object is an instance of a class, we have to ask, what class are classes instances of? The answer is that each class is the sole instance of its metaclass.

Using the message `class`, we can ask any object to what class it belongs. Thus, for example, the expression `5 class` produces the result `SmallInteger`. If we now use the expression

Fig. 11.15 Some classes and their metaclasses



SmallInteger class

to ask the class SmallInteger to what class *it* belongs, we receive the answer:

SmallInteger class

This expression represents the metaclass of SmallInteger; in other words, metaclasses do not have their own names.

With regard to class hierarchy the following applies: If class A is a subclass of class B, an analogous relationship holds true for their metaclasses. Figure 11.15 shows this relationship for a few selected classes of the class hierarchy.

There is though one difference between the hierarchy of the “ordinary” classes and that of their metaclasses. Although the class Object has no superclass (The expression `Object superclass` produces `nil`.), the expression

Object class superclass

produces the class named `Class` as the result, which is in turn an instance of its metaclass `Class class`. Figure 11.16 shows this connection.

And because metaclasses are also objects, we can again ask the question, what class are metaclasses instances of? The answer is: The class `Metaclass` is the class of all metaclasses. `Metaclass` itself is in turn an instance of its metaclass `Metaclass class`, which like all metaclasses is an instance of the class `Metaclass` (see Fig. 11.17). We could say that at this point, the circle closes in upon itself.

The system forms the basis for something that's characterised as *reflection* and *metaprogramming*. This refers to the basic ability to write programs that experience information about themselves and which are thus capable of changing. This is supported all the more by

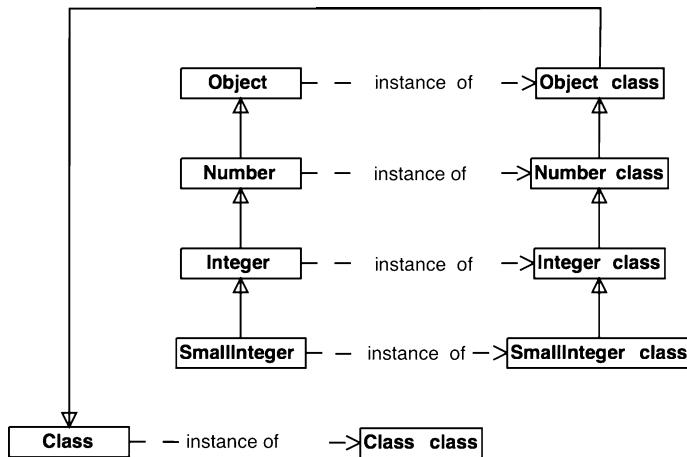


Fig. 11.16 Class is the superclass of Object class

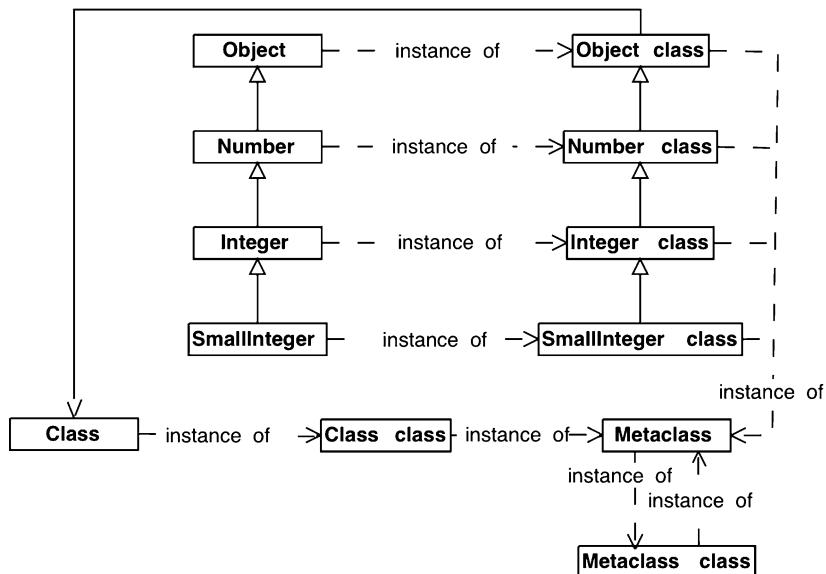


Fig. 11.17 All metaclasses are instances of Metaclass

the fact that all methods are accessible as objects at runtime. It's beyond the subject of this book to present all of the uses of metaprogramming. At this point, we'll just mention that these techniques made it possible to program in Smalltalk the tools of the Smalltalk development environment such as Compiler, Browser and Debugger. They appear in source code in the class library and can be changed, expanded or supplemented with additional tools. In the Smalltalk system Squeak, even the VM is programmed in Smalltalk.

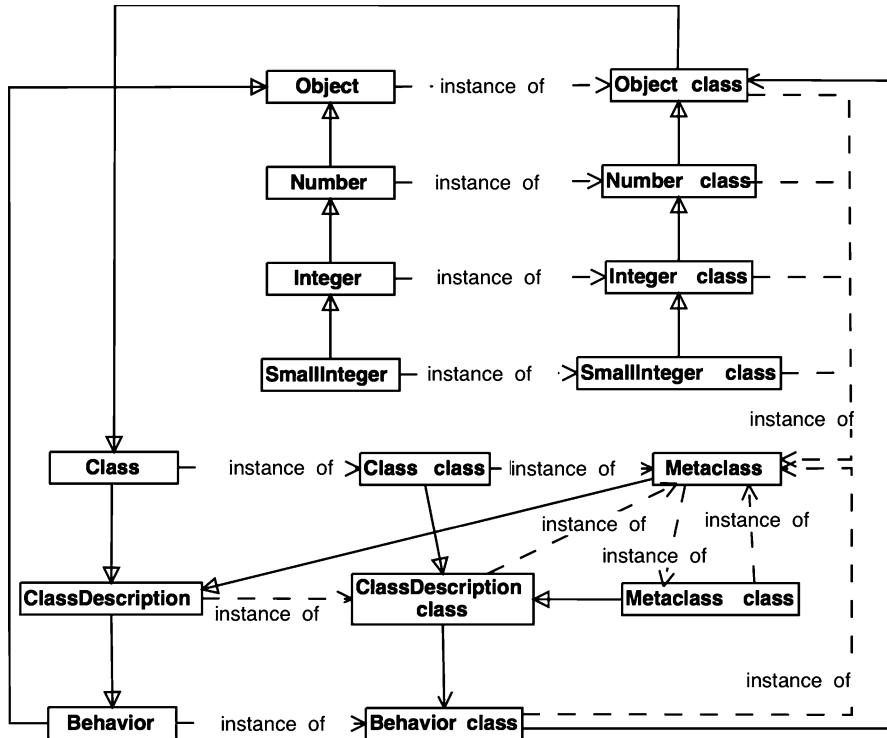


Fig. 11.18 Putting the classes `ClassDescription` and `Behaviour` into the diagram

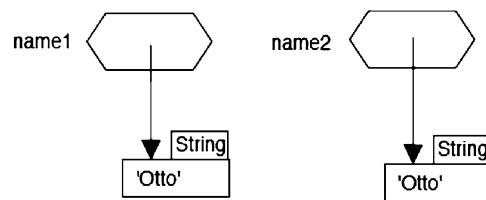
The class diagram shown in Fig. 11.17 is not entirely complete to the degree that it's missing the superclasses of `Class` and `Metaclass` as well as their metaclasses. Smalltalk's class hierarchy is like a tree with the class `Object` at the root, because all classes are either directly or indirectly subclasses of `Object`. Thus, in order to complete the diagram in Fig. 11.17, we must still add the “system classes” `ClassDescription` and `Behaviour` as well as their metaclasses; at the moment, we won't go any further into their meaning. Adding these classes produces the diagram shown in Fig. 11.18. Now we recognise that all classes that appear in this diagram are either directly or indirectly descended from `Object`.

11.4 Object Identity

Objects must be unambiguously identifiable in the memory of the VM. As a rule, this is accomplished by giving objects an unambiguous name. We usually refer to these object names as variables. By assigning an object to a variable, a bond is established between the object and its name. Each object name (each variable) can only ever be bound to exactly

Table 11.2 Testing the equality and identity of objects

Expression	Meaning
<code>obj1 = obj2</code>	Yields <code>true</code> when both objects are equal; otherwise it yields <code>false</code>
<code>obj1 ~= obj2</code>	Yields <code>true</code> when both objects are unequal; otherwise it yields <code>false</code>
<code>obj1 == obj2</code>	Yields <code>true</code> when both variables reference one and the same object; otherwise it yields <code>false</code>
<code>obj1 ~~ obj2</code>	Yields <code>true</code> when both variables reference different objects; otherwise it yields <code>false</code>

Fig. 11.19 Two equal but not identical String objects

one object. If an object is no longer bound to a name, it cannot be addressed; it loses its right to exist and is removed from object storage.

It is, however, entirely possible to bind more than one name to an object. We've already considered this fact in detail in Sect. 3.4. If two variables are bound to a common object, then they point to one and the same (identical) object. Stated somewhat simply, we say at this point that the variables are identical.

But two variables can point to two different objects that possess their own identity but which are equivalent in the sense that the values of all their instance variables agree. In that case, we say that the variables or the objects bound to them are equal.

11.4.1 Equality Versus Identity

We can see therefore that Smalltalk distinguishes between the identity and the equality of objects. This leads to the existence of different comparison operators, which are summarised in Table 11.2.

The following series of messages creates the situation shown in Fig. 11.19 in object storage:

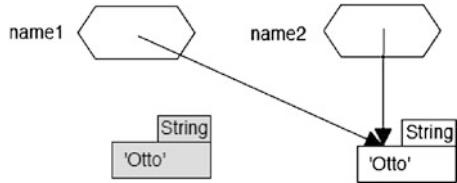
```

| name1 name2 |
name1 := 'Otto'
name2 := 'Otto'

```

Both variables refer to a `String` object with its own identity in object storage. Testing the variables for equality and identity produces the following result:

Fig. 11.20 name1 and name2 are identical



```

| name1 name2 |
name1 := 'Otto'
name2 := 'Otto'
name1 = name2. "Print it: true"
name1 == name2. "Print it: false"
  
```

Adding an additional assignment

```
name1 := name2
```

produces the situation shown in Fig. 11.20 in object storage. Now both tests

```

name1 = name2.
name1 == name2.
  
```

produce the value `true`.

At this point, we'll examine again the difference between `String` and `Symbol` objects that was mentioned in Sect. 10.2. Each symbol exists in object storage exactly once. That's why the following series of messages leads to the situation shown in Fig. 11.21:

```

| name1 name2 |
name1 := #Anna.
name2 := #Anna.
name1 == name2. "Print it: true"
  
```

When the test of two variables for identity produces `true`, that also applies, of course, to the test for equality.

If two variables point to the same object, we also say that they *share* the object, and we use the term *object sharing*. Because of the assignment semantics that Smalltalk uses (see Sect. 3.4), object sharing occurs frequently. That might mean that two different objects have identical components, which can easily lead to confusion and program errors.

Fig. 11.21 Equal symbols are always identical

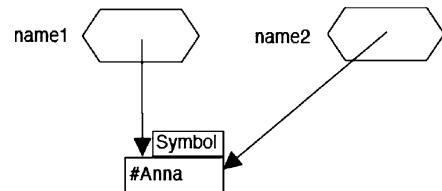
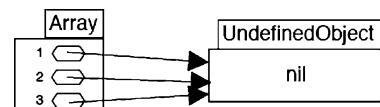


Fig. 11.22 An “empty” array with three components



For example, if we use the message `Array new: 3` to create an array with three elements, the three elements share the undefined object `nil` (see Fig. 11.22). This also occurs because the undefined object exists only once in object storage and cannot be duplicated.

Now let's look at the situation that occurs in object storage when we fill the array with values and then transform it into an `OrderedCollection`:

```

| aString anInteger aPoint anArray anOrderedCollection |
aString := 'Hallo'.
anInteger := 25.
aPoint := 203.
anArray := Array with: aString with: anInteger with:
aPoint.
anOrderedCollection := anArray asOrderedCollection.
  
```

In Fig. 11.23, we can see that, on the one hand, the three variables `aString`, `anInteger` and `aPoint` share their objects with the three elements of the array. On the other, we see that the message `asOrderedCollection` creates a new collection object, but that it contains the same components as the array. That means that the following two expressions both produce `true`:

```

anInteger == (anArray at: 2).
(anArray at: 2) == (anOrderedCollection at: 2)
  
```

But object sharing can prove troublesome if one of the objects involved is changed. If we add the message

```

aPoint x: 25
  
```

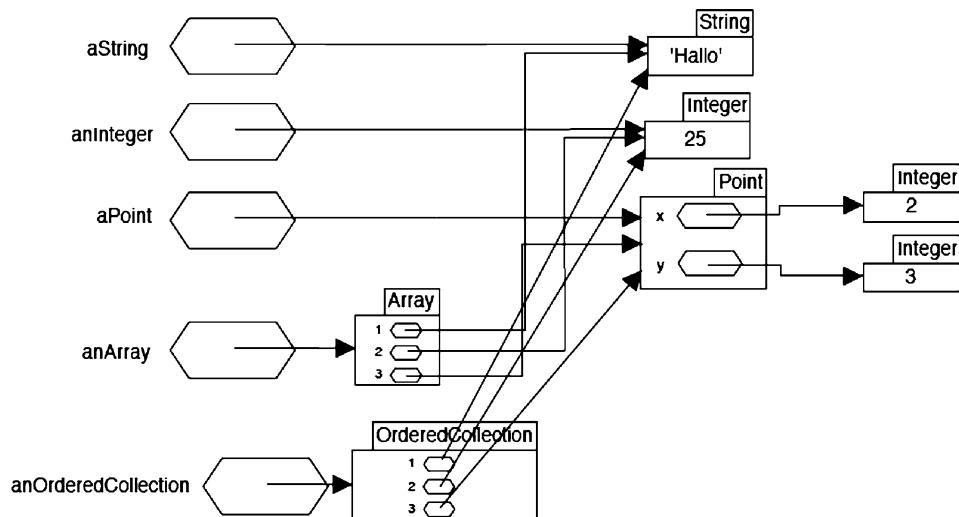


Fig. 11.23 The effect of *object sharing*

to the above sequence, then the `x` coordinate of `aPoint` takes on a new value. But the consequence of that is, say, that the expression

```
(anOrderedCollection at: 3) x
```

also produces the value 25. We find traces of the change in the `Point` object in all three existing assignments, because the `x:` message changes the condition of the existing `Point` object but does not create a new one. This is also sometimes called a side effect, and they are not always desirable. In such a case, it's possible to copy an object and then change it.

After the changes that we made to the `Point` object, the following two comparison expressions both produce true:

```
(anOrderedCollection at: 3) x = anInteger
(anOrderedCollection at: 3) x == anInteger
```

The `x` coordinate of the point bound to `(anOrderedCollection at: 3)`, like the variable `anInteger`, has the value 25. They're both also identical because all number objects are unique, that is, numbers do not have copies.

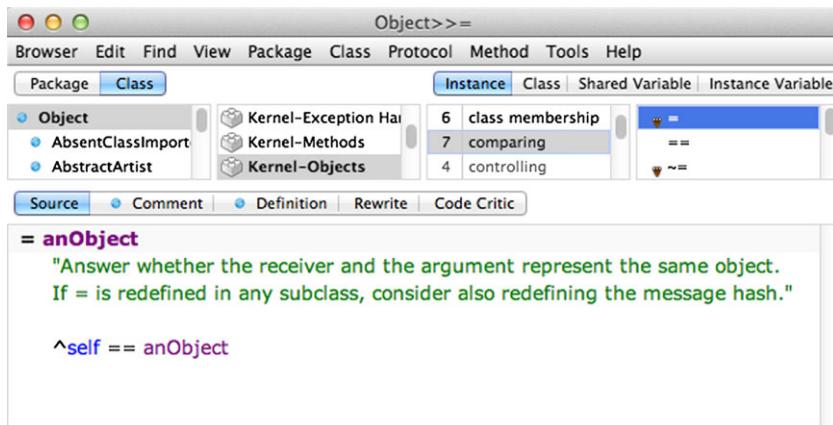


Fig. 11.24 The method “==” for the class Object

11.4.2 Equality of Objects of Self-Defined Classes

In the last section, we only looked at objects belonging to classes in the class library that comes with the product. What about the equality though of self-defined classes? Let's look again at the class `Person` that we defined in Sect. 11.2. If we now create two initialised `Person` objects and test them for equality, we obtain the following result:

```
| p1 p2 |
p1 := Person new.
p2 := Person new.
p1 = p2           "Print it: false"
```

It's true that both variables point to two `Person` objects that agree in all of their instance variables. But we haven't defined equality with respect to persons—in other words, we haven't defined a “==” method for the class `Person`. The test uses the method inherited from the class `Object`, and as Fig. 11.24 shows, that method tests for identity.

The programmer can arbitrarily define when two objects of a user-defined class should be regarded as equal. That's why it makes sense that the class `Object` can do nothing more than an identity test. For example, it's entirely conceivable that one might want to regard two `Person` objects as equal if they only agreed in certain specified instance variables but not necessarily all. That could then be accounted for in the definition of the method “==” in the class `Person`.

On the other hand, we can also ask ourselves in this case why copies of a `Person` object should even exist in object storage. That is, it's not imperative that we create a method to define equality for objects of a self-defined class.

If we do want to do it, the comment in the method “=” in the class `Object` (see Fig. 11.24) gives us a hint. In this case, we should also define the method `hash`.

Every object understands the message `hash`, that is, the class `Object` contains such a method. It implements a so-called *hash function*, which maps objects onto a subset of whole numbers. The hash function must satisfy the requirement that objects that are equal (`=`) are mapped onto the same whole number. As a rule, though, the definition scope of the function (the set of all objects) is greater than the scope of its values (subset of whole numbers), so that it can happen that two unequal objects may be mapped onto the same number.

Using hash functions is a frequently used programming technique for efficiently accessing data structures. In Smalltalk, for example, it is used to store elements in sets. When an element is added, we have to check whether the set already contains the same element. When one uses the hash value of an object to calculate the storage location within the data structure, one can quickly determine whether an element already exists equal to one about to be added. That is, if the location calculated from the hash value of the new element is empty, that means that no equivalent object is already in the data structure. The reverse case, though, is not as easy to deal with, since unequal objects may still have the same hash value. We won’t bother going into any further detail though. For Smalltalk programming, it’s enough to know that, if you want to define equality of instances for a user-defined class, you must first overwrite the inherited hash method. Otherwise it’s usually enough to make use of samples of hash methods that can be found in the class library.

As an example, let’s say that we wanted to base the equality of instances in our class `Person` on the equality of last names; in that case, we’d have to define the following instance method:

```
= aPerson
  ^self lastName = aPerson lastName
```

As an argument for the hash function, we then use only the instance variable `lastName`:

```
hash
  ^self lastName hash
```

We’re make use of the fact that a person’s last name is a character string, and that there’s a hash method defined in the class `String`.

With these additions, a comparison of two `Person` objects defined using `new` produces `true`:

```
| p1 p2 |
p1 := Person new.
p2 := Person new.
p1 = p2           "Print it: true"
```

If you want to use more than one component to test for equality, we would apply the hash function to a specially structured combination of components or else connect the hash values of the individual components. Let's assume that two people will only be regarded as equal if they have the same first and last names. Then we could start by implementing the following = method:

```
= aPerson
  ^self lastName = aPerson lastName
    and: [self firstName = aPerson firstName]
```

For example, we could decide to use the combined first and last names as an argument for the hash function:

```
hash
  ^(self lastName, self firstName) hash
```

Or, we could explore other hash functions that combine two instance variables of an object. One procedure used in the class library (in the class Fraction, for example) combines the hash values of the components on a bit-by-bit basis using exclusive or:

```
hash
  ^(self lastName hash) bitXor: (self firstName hash)
```

Without looking deeper into hash-function theory, it's impossible to determine which is the better option. That, though, is not what this book is about.

11.4.3 Object Copies

Connected to the theme of object identity is the question whether—and if so, how—objects can be copied. We've learned already that there are classes whose objects can't be copied. These include, among others:

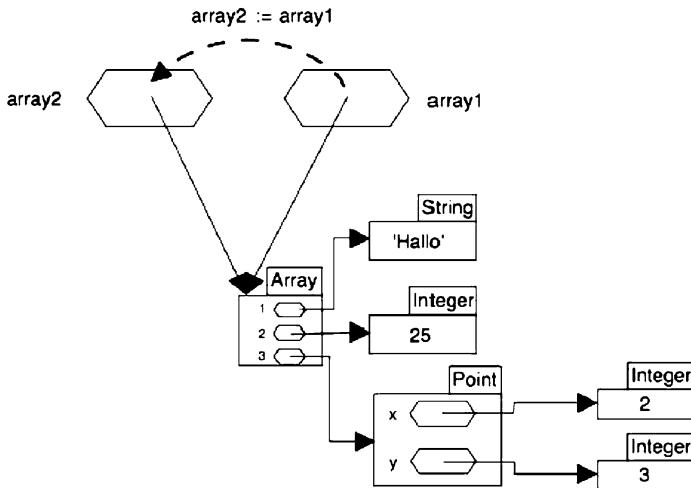


Fig. 11.25 Assigning an array to a different variable

- Number classes
- The class `UndefinedObject`, with the sole instance `nil`
- The classes `True` and `False`, with the only instances `true` and `false`
- The class `Symbol`

Each instance of each of these classes exists only once in object storage in the VM. But objects of other classes can certainly be copied. After making a copy, an object can be accessed via two different names (variables). The copy can occur at three levels:

1. *Variable assignment* simply copies references to objects; the object itself is not copied. Both variables refer to the same object.
2. A so-called *shallow copy* is a duplicate of the object, which shares the objects bound to the instance variables with the original object. The duplicate is equal to the original object (`=_`, but it is not identical to it (`~~`)). Nevertheless the objects bound to the instance variables of the objects and its duplicates are identical.
3. A so-called *deep copy* is a duplicate of the object with a shallow copy of the objects bound to the instance variables.

Let's use an `Array` object to examine these three variants. Figure 11.25 illustrates once again the effect of a simple assignment. The only thing copied here is the reference in the variable `array1`, which is copied to the variable `array2` so that afterwards both variables are bound to the one single array.

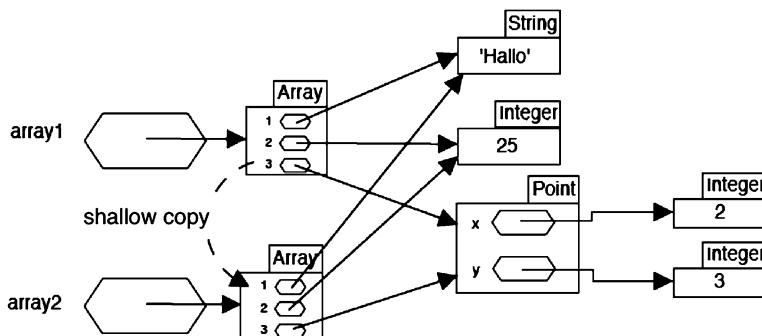


Fig. 11.26 Shallow copy of an array

Figure 11.26 shows the effect of a shallow copy. A second `Array` object is created and the references contained in the instance variables of the original array are copied to the corresponding instance variables of the new array. On the level of the instance variables, finally assignments again occur (copies of references).

We can create such a shallow copy in Smalltalk using the message `shallowCopy`. The process shown in Fig. 11.26 corresponds to the Smalltalk expression:

```
array2 := array1 shallowCopy
```

After finishing this shallow copy, the following expressions each produce `true`:

```
array1 = array2.
array1 ~~ array2.
(array1 at: 3) == (array2 at: 3)
```

A process similar to a shallow copy occurs when we send a collection one of the transformational messages discussed in Sect. 10.3, for example, `asOrderedCollection` (see Fig. 11.23). It is not a true copy, though, because of the transformation that occurs.

Finally, Fig. 11.27 shows the process of a deep copy. Like with the shallow copy, a second `Array` object is created. In addition though, shallow copies of the objects bound to the source array are created. Nevertheless, no copy is made of the second array component (the integer 25) because, as has already been stated, number objects are unique.

The class library in VisualWorks does not contain a method for deep copies. Instead, the class `Object` contains—besides the `shallowCopy` method—the following method:

```
copy
  ^self shallowCopy postCopy
```

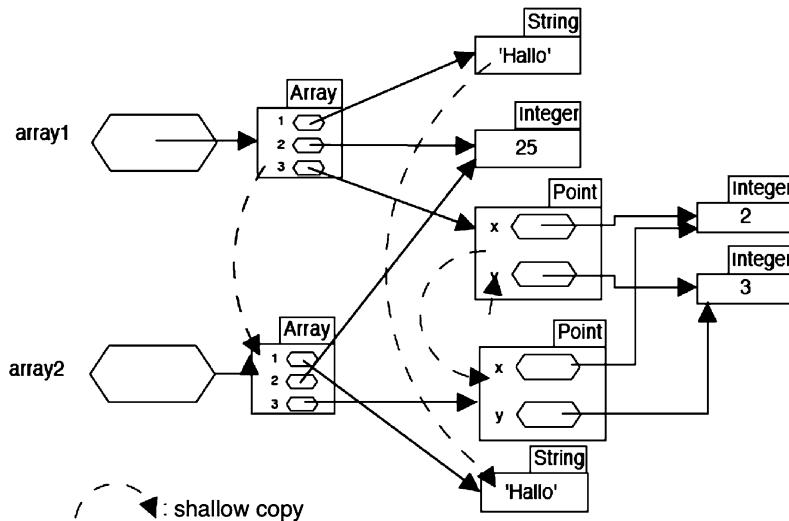


Fig. 11.27 Deep copy of an array

In this case, `shallowCopy` creates a shallow copy and then sends the copy the message `postCopy`. The method `postCopy`, defined in the class `Object`, does nothing but return the receiver. Nevertheless, one can overwrite the method `postCopy` in a user-defined class and thereby create a deep copy.

The class library of VisualWorks does not contain many classes that implement the message `postCopy`. One of them is `Rectangle`, in which the method is defined as follows:

```
postCopy
super postCopy
origin := origin copy.
corner := corner copy
```

An instance of the class `Rectangle` possesses the two instance variables `origin` and `corner` (see Sect. 5.7), which form the endpoints of the diagonal of a paraxial rectangle. It is a convention of a `postCopy` method to activate before anything else the method in the superclass so that inherited instance variables can be copied. In this case, though, there is no superclass for `Rectangle` other than `Object`, in which the `postCopy` method is implemented. The expression `super postCopy` thus has no effect here. The next two lines then use `copy` to create copies of the `Point` objects stored in the two instance variables.

Whether we're dealing with a shallow or a deep copy depends on whether `postCopy` is implemented in the class `Point`; however, it is not. It wouldn't make any sense if it were because a `Point` object simply refers to two whole numbers that can't be copied anyway. After executing the following series of messages

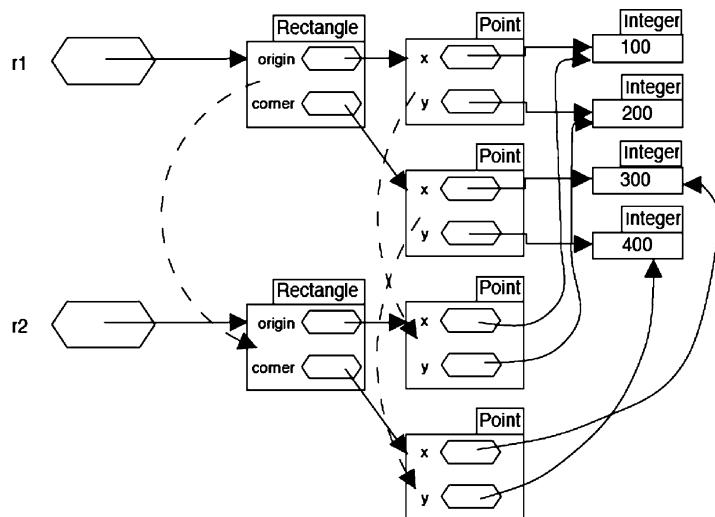


Fig. 11.28 Deep copy of a Rectangle object

```
| r1 r2 |
r1 := Rectangle origin: 100@200 corner: 300@400.
r2 := r1 copy.
```

the following expressions produce true:

```
r1 ~~ r2.
r1 origin ~~ r2 origin.
```

Figure 11.28 illustrates the procedure for copying the rectangle.

Deep vs. Shallow Copies

You may wonder when shallow or deep copies are preferable. In general terms, shallow copies require less effort and are therefore preferable, unless other concerns argue for deep copies. The problem with shallow copies is that it is possible for side effects to appear, since the original and the copy share the objects bound to their instance variables. A change in the status of one of these objects necessarily affects both the original and the copy. If a shallow copy of a rectangle were created, changing the `origin` or `corner` components in the copy would also affect the original; in other words, both rectangles would change. As a general rule, that would be an undesirable effect for a rectangle.

Deep copies are, in general, preferable if the components of an object represent irreducible parts of the whole—such as, for example, the endpoints of the diagonal of a rectangle—and which might in and of themselves have no independent existence. Another

example might be an object that represents a book. The objects bound to the instance variables of which are the chapters of the book. It doesn't seem to make much sense to create a copy of the book object without also copying the chapter objects. If, however, another instance variable of the book object refers to the author of the book, then when a copy of the book is created, the object referring to the author cannot under any circumstances be copied. Otherwise, two objects would exist in the memory of the VM representing the exact same object from the real world.

Shallow copies do not present a problem when side effects cannot occur because the objects that the original and the copy share cannot be changed. First, this applies to all objects that cannot be copied. In this case, the behaviour of shallow and deep copies is the same. In Figs. 11.26 and 11.27, we can see this in the second element of `array1` and `array2`, which refer to the same integer object in both the shallow and the deep copy. Then, since version 7 of VisualWorks String objects are also among the objects that cannot be changed, that means that expressions like

```
aString at: anInteger put: aCharacter
```

cause an exception. Thus it is possible to use the expression

```
array2 at:2 put: 'Hello'
```

to assign a new `String` object to the second element of `array2` in Fig. 11.26. This would have no effect though on the second element in `array1`.

In this chapter, we'll return to a topic that we spent a lot of time on earlier, especially in Chaps. 2 and 4. In those chapters, we considered basic principles and means for designing algorithms. In addition, in Sect. 11.1.3, we hinted at a description of how messages are implemented through the use of *recursion*.

Recursion is among the most important design principles that computer science uses for algorithms and data structures. In general, recursive structures are those that contain themselves as components. Since this abstract description probably serves to obscure rather than to enlighten matters, let's use examples to approach a better understanding of the recursion principle.

Computer science didn't invent recursion; we can find examples of recursive definitions throughout mathematics too.

We can define natural numbers in the following way:

- 1 is a natural number
- The successor of a natural number is also a natural number.

Put simply, this is a recursive definition because the concept we're defining is used in the second part of the definition. The definition implies that the set of natural numbers is infinitely large, because we can say that the next number after every natural number is itself another natural number.

Another well-known example is the definition of the factorial of a natural number n , which is written as $n!$:

$$n! = \begin{cases} n = 1 : 1 \\ \text{else: } (n - 1)! \cdot n \end{cases}$$

An important feature of this definition is that it contains a branching decision. A recursion occurs only for the case where $n \neq 1$. At the same time, the definition provides calculation instructions, which tell us to calculate $4!$ as follows:

$$\begin{aligned}
 4! &= 3! \cdot 4 \\
 &= 2! \cdot 3 \cdot 4 \\
 &= 1! \cdot 2 \cdot 3 \cdot 4 \\
 &= 1 \cdot 2 \cdot 3 \cdot 4 \\
 &= 24
 \end{aligned}$$

The recursion ends in the penultimate line, because the result for $n = 1$ can be written directly.

In Sect. 11.1.3, we learned about recursive blocks. We can transfer the above recursive definition of a factorial into the following recursive block:

```

| fac |
fac := [:n|
  (n=1)
    ifTrue: [1]
    iffFalse: [(fac value: n-1) * n]].
fac value: 4 "Print it: 24"

```

Within the definition of the block `fac`, the message `value: n-1` is sent to itself in the `iffFalse:` branch. That is, it is used within its own definition. In other words, we have been presented with a recursive algorithm.

12.1 Recursive Algorithms

The recursive evaluation of the block in the algorithm for calculating a factorial shown above causes a specific part of the calculation to be repeated. Recursion thus provides us with an additional algorithmic means of expression for parts of an algorithm that are to be repeated. In Chap. 4, we already learned about a variety of other options for expressing repetitions. Such repetitions are called *iterations*, and the corresponding algorithms are termed *iterative*. Before discussing matters any further, we'll start with the following definition of a recursive algorithm:

We call an algorithm *recursive* when one of the elementary components that define it is the algorithm itself. Otherwise we call it *iterative*. An algorithm is called *directly recursive* if it serves as a component in its own definition. It's called *indirectly recursive* if it serves as a component in the definition of one of its components.

In both mathematics and computer science, a recursive solution can often be derived very elegantly from a problem statement. But for each recursive algorithm, it's possible to provide an equivalent iterative algorithm and vice versa. In some cases, where there's an

obvious recursive solution for a problem statement, the iterative solution can sometimes be more complicated and confusing.

Nevertheless, the hardware available for typical processors only allows the writing of iterative programs; that is, at least by the time the program gets to the compiler or the interpreter, a recursive algorithm will be turned into an iterative one. That's also the reason why iterative programs tend to be faster than recursive ones, which means that it's best to avoid unnecessary recursivity.

A distinctive feature of Smalltalk is certainly the fact that the messages in the class library that are available for iterations are derived from recursive implementations (see Sect. 11.1.3). But we can assume that the compiler has created a very efficient iterative code for the VM.

As we've already mentioned, some algorithms have a natural recursive formulation. Clarity and legibility along with maintainability represent important criteria for the quality of algorithms; at times, they may be as valuable—or even more valuable—than efficiency. An additional important aspect is the provability of an algorithm. Recursive formulations of an algorithm are often very suitable for proving correctness by means of complete induction.

Unnecessary Recursion: Factorial

The algorithm for calculating a factorial that we studied earlier is an example of unnecessary recursion, because it could be formulated naturally in an iterative manner. This time, we'll define a method that we'll place in the class `Integer`:

```
factorialIterative
    "calculates the factorial of the receiver in an
     iterative process. It is assumed that
     the receiver is greater than or equal to 1."
    | tmp |
    tmp := 1.
    2 to: self do: [:i | tmp := tmp * i].
    ^tmp
```

So that we can have an idea that the iterative solution is less complex, we'll now define a corresponding recursive method:

```
factorialRecursive
    "calculates the factorial of the receiver in a
     recursive process. It is assumed that
     the receiver is greater than or equal to 1."
```

```

self = 1
ifTrue: [^1]
ifFalse: [^self * (self - 1) factorialRecursive]

```

In order to calculate $n!$ by the recursive algorithm, the following things need to occur:

- n activations of the method `factorialRecursive`
- n executions of the return operator
- n multiplications
- n conditional tests
- $n - 1$ subtractions

The following effort is required for the iterative formulation:

- 1 activation of the method `factorialIterative`
- $n - 1$ multiplications
- n associations
- 1 execution of the return operator
- $n - 1$ loop administrations

On most computers or VMs, the iterative process is likely to run faster than the recursive. That's greatly dependent, though, on how well the computer is able to optimise the recursive method. In the class `Time` in the Smalltalk class library, the class method `microsecondsToRun:` determines in microseconds the computing time required for the evaluation of the block sent as an argument. As an example, we'll use that method to determine $100!$. If we repeat the process 1000 times

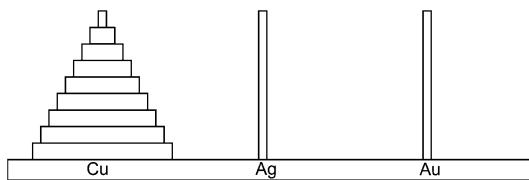
```

(Time microsecondsToRun:
[1000 timesRepeat:
[100 factorialIterative]]) //1000 "Print it: 31"
(Time microsecondsToRun:
[1000 timesRepeat:
[100 factorialRecursive]]) //1000 "Print it: 40"

```

there's no significant difference between the average calculation times. But this result is not representative for the various brands of compilers, VMs or even programming languages.

Both processes though are—as we said—on the order of n . That means that n steps are required for the execution of the algorithm, although the complexity of the individual steps is ignored.

Fig. 12.1 The towers of Hanoi

On occasion, it makes sense from an efficiency point of view to consider an entirely different solution to calculating a factorial: It might be that an application requires only factorials for numbers between 1 and an upper limit of m . For that reason, it's much simpler to create a table with factorials between 0 and m , that is, to create an array named `factorials`. With that preparation, we calculate $n!$ using the Smalltalk expression:

```
factorials at: n
```

This algorithm is on the order of 1, but—depending on m —might cost a lot more memory. This phenomenon—being able to optimise either computer time or memory only at the expense of the other—occurs quite frequently while developing algorithms.

The Towers of Hanoi

A lovely—though otherwise useless—example of an algorithm that can be formulated recursively with astonishing simplicity is based on a legend that computer scientists have been telling for a long time, and which obviously exists in many versions: the legend of the towers of Hanoi. I heard the legend a long time ago in approximately these words: Once upon a time three columns stood in front of a temple in Hanoi, one made of copper, another of silver and a third of gold. On the copper column, there were one hundred porphyry disks, each one smaller than the one beneath it (see Fig. 12.1).

An elderly monk was then given the following task:

Move the tower with the disks from the copper to the golden column. In each case, take the topmost disk from one column and place it on top of another column in such a way that a larger disk is never placed on top of a smaller one.

The legend goes on to say that, once the monk had completed his task, the end of the world would be at hand.

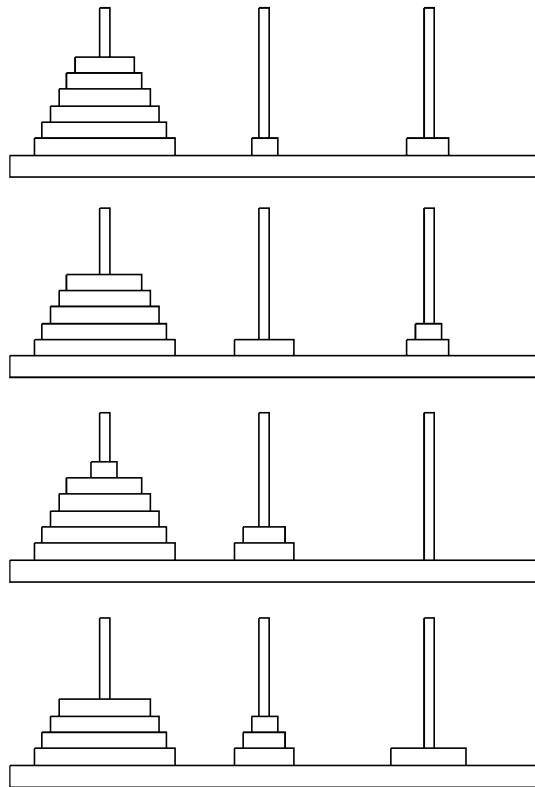
In order to illustrate the monk's difficult task, Fig. 12.2 shows how the first disks were restacked.

After intensive consideration, the old monk had two insights:

1. In order to complete the task, the silver column must also be used.
2. The task can be solved in essentially three steps.

Step 1: Move the first 99 disks of the tower from the copper to the silver column.

Fig. 12.2 Restacking the first disks of the towers of Hanoi



Step 2: Move the last, largest disk from the copper to the golden column.

Step 3: Finally, move the tower of 99 disks from the silver to the golden column.

As the monk reviewed this procedure, he realised that Steps 1 and 3 were too difficult for him. Therefore, he decided to let his oldest student perform Step 1. After he finished his work, the old monk would himself move the large disk from the copper to the golden column, after which he would again make use of his oldest student.

But because the monk was not only old and wise but also fair, he told his brilliant plan to his oldest student, so that he too could make use of it in his task of transporting the 99 disks. Therefore, he generalised his plan in the following way:

The process for transporting a tower consisting of n disks from one column to another while making use of the third column:

Step 1: If the tower consists of more than a single disk, then ask your oldest student to transport $n - 1$ disks from the first column to the third, making use of another column.

Step 2: Carry one disk yourself from one disk to another.

Step 3: If the tower consists of more than one disk, then ask your oldest student to carry a tower of $n - 1$ disks from the third column to the other column while using another column.

The restacking of the towers thus begins with the old monk asking his oldest student to move the tower with the 99 uppermost disks. Now the reader should answer the following questions about this algorithm:

- What is the first thing that the student now does?
- How many monks and/or students will be involved in the work before the first disk is moved?
- What does the work of the n th monk look like?
- What does the old monk do when his student appears at his door?

Now we'll describe the old monk's algorithm in Smalltalk but, instead of moving disks, we'll just send an output to Transcript saying which disk we'll now move from which column to another. For that purpose, we'll define a method `transportTowerFrom:to:with:` for the class `Integer`, in which the receiver of the message indicates the height of the tower. The old monk's task will then be solved using the following Smalltalk expression:

```
100 transportTowerFrom: 'Copper' to: 'Gold' with: 'Silver'
```

The method is:

```
transportTowerFrom: one to: another with: aThird  
"...realizes the algorithm 'Towers of Hanoi'.  
Receiver of the message is a whole number that  
indicates the height of the tower."  
  
"Step 1:"  
self > 1  
ifTrue:  
    [self - 1  
        transportTowerFrom: one  
        to: aThird  
        with: another].  
  
"Step 2:"  
Transcript  
show: 'disk';
```

```

        show: self printString;
        show: ' from ';
        show: one;
        show: ' to ';
        show: another;
        cr.
    "Step 3:"
    self > 1
        ifTrue:
            [self - 1
                transportTowerFrom: aThird
                to: another
                with: one]

```

This is, in fact, a very simple and correct method, which can easily convince us that it works correctly, but which is extremely difficult for those not familiar with a recursive way of thinking. It's especially hard to see without further explanation how the statement can be true that the algorithm actually follows the rule that a large disk will never be placed on top of a smaller. In the next section, we'll sketch out a proof to show that this statement is actually valid.

For now, we'll be content with testing the algorithm by using a small tower, so that we can perform “by hand” the disk movements in the protocol and test them. The following is the protocol of the disk movements from the Transcript that occurs when one sends the message

```
4 transportTowerFrom: 'Copper' to: 'Gold' with: 'Silver'
```

in order to move four disks:

```

Disk 1 from copper to silver
Disk 2 from copper to gold
Disk 1 from silver to gold
Disk 3 from copper to silver
Disk 1 from gold to copper
Disk 2 from gold to silver
Disk 1 from copper to silver
Disk 4 from copper to gold
Disk 1 from silver to gold
Disk 2 from silver to copper

```

```
Disk 1 from gold to copper
Disk 3 from silver to gold
Disk 1 from copper to silver
Disk 2 from copper to gold
Disk 1 from silver to gold
```

And now we come to a remark in the legend, where it was said that the end of the world would occur when the monk had completed his task. And so we ask ourselves, how long will it take to move a tower consisting of 100 disks? First we'll try to determine how many disk movements would be required to move them all. It's easy to see that 7 moves would be necessary to move 3 disks; for 4 disks, 15 moves (see above) and for 5 disks, 31 moves. In general, it would appear that $2^n - 1$ disk movements are necessary to move a tower consisting of n disks. For a tower of 100 disks, that would be 1,267,650,600,228,229,401,496,703,205,375.

If the monks are very quick and move one disk every second, the entire task would take about $4 \cdot 10^{22}$ years.

12.2 Correctness of Recursive Algorithms

With recursive algorithms, one can try to prove correctness using a kind of *recursive induction*. If the induction variable for proving an iterative method is the number of times a particular point in the algorithm is run, then the variable for recursive induction is the *recursion depth* that causes a recursive call. A definition of recursion depth is:

When no recursive call occurs when a call is executed, then the recursion depth is 0; otherwise it is greater by 1 than the largest recursion depth of all subsequent calls that this call causes.

For example, the call

```
1 transportTowerFrom: 'copper' to: 'gold' with: 'silver'
```

has a recursion depth of 0 because no additional recursive calls occur. The call

```
2 transportTowerFrom: 'copper' to: 'gold' with: 'silver'
```

on the other hand causes two additional calls, each with a recursion depth of 0. It thus has a recursion depth of $0 + 1 = 1$.

In general, the call

```
n transportTowerFrom: 'c1' to: 'c2' with: 'c3'
```

has a recursion depth of $n - 1$. We could prove this again using complete induction, but we'll omit that at this point.

Using recursive induction in a proof proceeds as follows:

1. Formulate the statement that must be true for the algorithm.
2. Show that the statement is true for a call with a recursion depth of 0.
3. Show that the truth of the statement for calls with a recursion depth of $n + 1$ follows from the truth of the statement for calls with a recursion depth equal to n .

Correctness of the Algorithm *The Towers of Hanoi*

As a basis for the proof, we will use the method `transportTowerFrom:to:with:..`.

Statement: When executing the call

```
k transportTowerFrom: c1 to: c2 with: c3
```

none of the k disks will ever be set upon a smaller disk, and afterwards the upper k disks will have been moved from column $c1$ to column $c2$. The recursion depth is $k - 1$ (see above).

Anchor: If the recursion depth $k - 1 = 0$, exactly one disk will be moved to the column where the smallest $k - 1$ disks are not lying, that is, the statement for the recursion depth 0.

Inductive assumption: The statement is true for the recursion depth $k - 1 = n$, that is, $k = n + 1$.

Inductive conclusion: Now we examine the call

```
n+2 transportTowerFrom: c1 to: c2 with: c3
```

This call will be executed as

Step 1: `n+1 transportTowerFrom: c1 to: c3 with: c2`

Step 2: Transport a disk from $c1$ to $c2$.

Step 3: `n+1 transportTowerFrom: c3 to: c2 with: c1`

In Steps 1 and 3, calls with a recursion depth of n are made, for which the inductive assumption says the statement is true. In Step 2, the disk $n + 2$ (the biggest, if we assume that the disks on the tower are consecutively numbered from top to bottom

starting with 1) is moved from column $c1$ to column $c2$; as a result of Step 1, the $n + 1$ smaller disks are on column $c3$, which means that the statement is true.

Therefore we have used recursive induction to prove the statement for every recursion depth greater than or equal to 0, that is, for every disk greater than or equal to 1.

In order to completely prove the correctness of a recursive algorithm, we must also prove its *termination*. An algorithm terminates when it ends after a finite number of steps. For the algorithm Towers of Hanoi, this results fairly easily from the consideration that the recursion depth can never be less than 0, because—in a case where it is 0—no further recursive call occurs.

12.3 Recursive Thinking

One question that beginning programmers in particular tend to ask is, how do we find recursive solutions to a problem? In more general terms, the question is, how do we find any solution to a problem? The two simple questions unfortunately have no simple answers. There just are no patent recipes that lead from a stated problem to a correct algorithm. With regard to recursive algorithms though, we present here a few basic comments about the way of thinking that usually underlies them.

Recursive algorithms are frequently based on a “divide and conquer” strategy. In other words, recursive solutions offer themselves especially when a problem can be solved by dividing the original problem into smaller problems, and then knitting the solutions together into a solution to the entire problem.

With regard to the examples we examined in Sect. 12.1 though, we can make concrete the abstract thought we considered above. Solve a problem of “size” n on the assumption that there’s already a solution for the problem size $n - 1$. The recursive method `factorialRecursive` calculates $n!$ on the assumption that the value of $(n - 1)!$ already exists. The problem of moving a tower consisting of n disks according to the rules that apply to the Towers of Hanoi legend is solved on the assumption that the problem has already been solved for a tower with a height $n - 1$.

In each of these examples, each recursive call reduces the problem size by 1. For each complete recursive solution, it is expected that you be able to directly supply the solution for the smallest size (for example, $n = 1$).

To illustrate this way of thinking with another example, let’s look at a task of calculating the sum of a series of numbers stored in an array. For example, the expression

```
#(3 7 11 25) sum
```

should produce the result 46. How can we take the problem of calculating the sum of an array of the size n and relate it to calculating the sum of an array with $n - 1$ elements?

Well, if we know the sum of the first $n - 1$ elements, for the total sum, we need only add the n th element.

What does the “smallest” problem size look like in this case? If the array is empty, let the sum of the elements be 0.

These ideas lead to the following recursive method, which we’ll place in the class `Array` for simplicity’s sake:

```
sum
  "calculates recursively the sum of the elements
   of the receiver array"

^self size = 0
  ifTrue: [0]
  iffFalse: [(self copyFrom: 1 to: self size - 1) sum
             + (self at: self size)]
```

In the `iffFalse:` branch, the method is called with a copy of the receiver reduced by one element, after which the final element is added.

As we said, this example is intended only to illustrate a recursive way of thinking. Otherwise, there’s no reason to solve this problem recursively.

12.4 Infinite Structures

In this section, we’ll suggest the options that result on the one hand from the recursion principle and on the other from the fact that blocks are objects. As objects, blocks can, of course, also appear as components in a collection. Because blocks can appear beside other “data” in one and the same data structure, the difference between programs and data gets a little murky here (see Sect. 11.1.3 as well).

At this point, we can’t explore the full range of possibilities arising from this fact. Instead, we’ll use a simple example to give an impression of these possibilities.

Let’s assume that we need a data structure that contains an infinitely long, uninterrupted series of whole numbers beginning with a starting value to be provided. We can use a Smalltalk expression in the format

```
n from
```

to produce such a series, as an `OrderedCollection` perhaps, with a starting value of `n`. We define the method `from` in the class `Integer`. We know, of course, that no real-world memory could contain an infinitely long list, and so we’ll distance ourselves

for the moment from such technical limitations. Naively, we write the following recursive method `from`:

```
from
| list |
list := OrderedCollection new.
list add: self.
list add: (self + 1) from.
^list
```

If we were to actually activate the method, evaluating for example `1 from`, it would begin an endless process by first writing its receiver at the start of an `OrderedCollection`; then it would call itself recursively with a receiver increased by one. The resulting structure would then look like this:

```
OrderedCollection(1
    OrderedCollection(2
        OrderedCollection
            (3...))
)
```

Since the recursion does not end, the method is practically useless. For that reason, we'll make the following change:

```
from
| list |
list := OrderedCollection new.
list add: self.
list add: [(self + 1) from].
^list
```

The only difference—but a substantial one—is that the recursive call has been written as a block between square brackets. The `OrderedCollection` resulting from calling `from` now contains two elements. In the first, is the receiver, that is, the first member of the series of numbers; in the second, is a block that contains the calculation instruction for the next step. Thus, for example, evaluating `25 from:` produces:

```
OrderedCollection (25 BlockClosure [] in Integer>>from)
```

We have now built a structure, the first element of which is a “simple” number object—we might say that it’s data in the classic sense—with a second element that is a block—a piece of programming code.

Whenever the second member of the series is needed, the user of this “infinite” series must see to it that the block is evaluated. Let’s try to apply it to a method `sum:`, which will calculate the first n members of the number series sent as an argument. The number n is the receiver of the message. In order to calculate the first 100 members of a number series beginning with 25, we must evaluate the following expression:

```
100 sum: 25 from
```

Again we define the method `sum:` in the class `Integer`:

```
sum: list
^self = 0
    ifTrue: [0]
    ifFalse: [list first
              + (self - 1 sum: (list at: 2) value)]
```

The recursive principle (see Sect. 12.3) of this method again consists in the fact that the sum is determined by adding the first member of the series (`list first`) to the sum of the remaining members of the series. We determine the remaining members of the series, in this case, by sending the message `value` to the block that is the second element in the `OrderedCollection` `list`. That means that when `from` is called again, precisely the next element and the calculation instruction for the one after that are determined and then stored again in an `OrderedCollection`.

This makes clear the difference between the first, unusable variation of the method `from` and the second version. The recursive call that is present in the first variation is delayed in the second (using block brackets to cause a delayed evaluation) until it is needed, as is the case, for example, in the method `sum`.

Of course, the problem that we’ve presented here can be solved in a simple and efficient manner, without resorting to recursion or delayed evaluation. We can’t delve any deeper into the programming possibilities that the combination of these two principles create. You can find additional information in texts on the subject of *functional programming*, such as Pepper and Hofstedt (2006).

The messages for processing ordered collections that we dealt with in Chap. 10 require that the elements in a collection be processed without interruption. All operations that are to be executed on the elements must be gathered together into a block, which is passed to the collection as a parameter along with the relevant message. Sometimes though, it's more practical to be able to interrupt the processing of the collection elements and pick them up again at a later time, that is, at a different point in the program.

For this purpose, it would be very helpful if we could send the collection a message of the type

“deliver the next element”

We call this *sequentially accessing* the collection. In order to be able to do this though, the collection object at least must be able to keep track of which element was last accessed. No provision for that action exists in the collection classes.

Instead, Smalltalk has the capability of “jamming” an administrative structure onto an ordered collection. These administrative structures are instances of the class `Stream` or of its subclasses `ReadStream` and `WriteStream`. We can say that a `ReadStream` connected to an ordered collection transforms the collection into a continuous stream that had been processed sequentially from start to finish. The `ReadStream` object controls which object is to be produced at the next access. Section 13.1 contains additional details.

It happens very often that application programs have to process data that is stored in files on external storage media. These files might be text, images or videos. The easiest and most common way to process the data contents is to access them sequentially. That fact prompted the designers of the Smalltalk class library to regard files too as external streams that can be processed with similar messages to those used on internal streams connected to collections. Section 13.2 deals with sequential access to files.

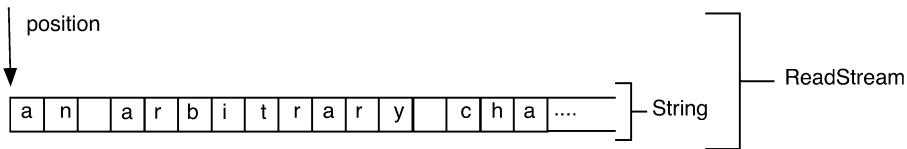


Fig. 13.1 Linking a ReadStream to a String

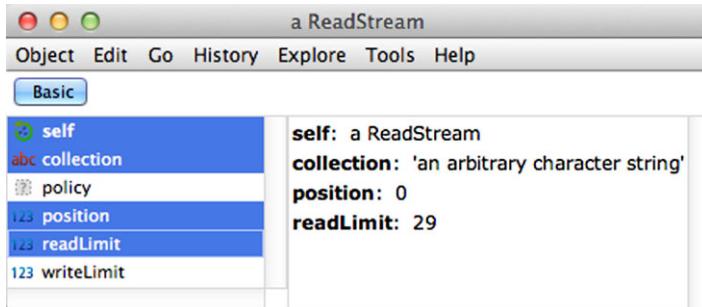


Fig. 13.2 A ReadStream object

13.1 Sequential Access to Ordered Collections

We've seen that the class `Stream` allows sequential access to ordered collections within Smalltalk. The important subclasses for application programmers are `ReadStream` and `WriteStream`. As a rule, streams are processed either by being read or by being written to. Let's look at read access first.

Read Access

Our starting point is an ordered collection that's full of data that is to be processed sequentially. The collection should be connected to a `ReadStream` object, which can happen in the following way:

```
| string readStream |
string := 'an arbitrary character string'.
readStream := ReadStream on: string.
```

In this case, the collection is a `String` object; the message `on:`, sent to the class `ReadStream`, links the collection, which is transmitted as a parameter, to an instance of this class. Figure 13.1 shows a schematic representation of this.

In an instance variable called `position`, a `ReadStream` object controls the location of the next object to be sequentially accessed—and read—from the collection. If we use **Inspect it** to execute the above sequence, we see (as shown in Fig. 13.2) that, once the `ReadStream` instance is created, the variable `position` has the value 0.

The message `next` executes a read access to a `ReadStream` object. If we add

```
character := readStream next.
```

to the above series of instructions, the index `position+1` accesses the `String` object and the character `$a` is put in the variable `character`. Next, the value of the instance variable `position` is increased by 1, so that the subsequent execution of `next` produces the character `$n`.

Figure 13.2 also shows that a `ReadStream` object keeps track of the number of components in the linked collection in the instance variable `readLimit`. An attempt to use the message `next` to access the 30th character would produce an exception. You can use the message `atEnd` to test whether the `ReadStream` contains additional unread objects. The following Smalltalk sequence shows an application; it rights all to consonants in the character string to Transcript:

```
| string readStream |
string := 'an arbitrary character string'.
readStream := ReadStream on: string.
[readStream atEnd]
  whileFalse:
    [ | character |
      character := readStream next.
      character isVowel ifFalse: [Transcript
                                    nextPut: character]]
```

Write Access

For write access, a `WriteStream` object is linked with an ordered collection, which is usually empty at the start. As before, the process uses the message `on:`. Using the message `nextPut:`—sent to the `WriteStream` object—an object can be written to the end of the collection. The instance variable `position` always contains the index of the most recently written object. The Smalltalk sequence

```
| array writeStream |
array := Array new.
writeStream := WriteStream on: array.
1 to: 10 do: [:i | writeStream nextPut: i]
```

writes the numbers from one to 10 into the array. In connection with `WriteStream`, it in fact makes sense to create an empty array object, since no objects can be added to an empty array using messages for ordered collections (see Sect. 10.2).

Using the message `nextPutAll:` you can write the contents of the collection sent as an argument to the end of the collection linked to the `WriteStream`. This also means that you have a very efficient way to combine two collections, as the following instructions show:

```
| string stream |
string := String new.
stream := WriteStream on: string.
stream nextPutAll: 'an arbitrary'.
stream nextPutAll: ' character string'.
stream contents
    " Print it produces: 'an arbitrary character string'"
```

The message `contents` produces a copy of the collection linked to a `Stream` object from index range 1 to `readLimit`. Two character strings can also be chained using the binary message “`,`”. In that case though, both operand character strings are copied into memory. This can be an especially time-wasting process if, for example, one of the character strings is lengthened within a loop every time the procedure runs. In this case, we recommend using `Stream` objects, which make use of particularly efficient memory management.

We already learned about the messages `nextPut:` and `nextPutAll:` in Sect. 5.3.3. They also serve the purpose of writing individual characters or character strings to the Transcript. The Transcript is an instance of the class `TextCollector`, and its instance variable `entryStream` points to a `WriteStream` object. A `nextPut:` or `nextPutAll:` message sent to the Transcript will ultimately be forwarded to this object.

`WriteStreams` are also used for implementing `printOn:` methods. We’ll return to this in Sect. 14.2.

Common Messages for `ReadStreams` and `WriteStreams`

The class `PositionableStream` is the common abstract superclass for (among other classes) `ReadStream` and `WriteStream` (see Fig. 13.3). In the protocols `accessing`, `positioning` and `testing`, it provides some useful messages that can be used for both `ReadStreams` and `WriteStreams`, and which are here described in part.

We’ve already explained the messages `atEnd` and `contents`. You can use `isEmpty` and `isNotEmpty` to determine whether a `Stream` or the collection linked to it are empty.

The messages `position` and `position:` allow the querying and placement of the instance variables with the same names; for the latter, no value less than 0 or greater than `readLimit` can be specified. The message `reset` sets `position` to 0. The message

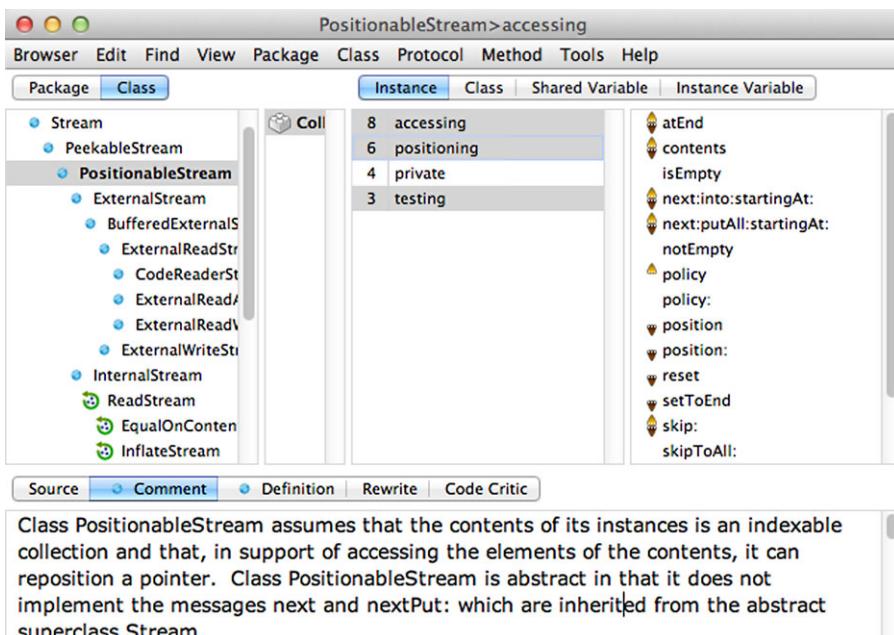


Fig. 13.3 The class PositionableStream

`skip:` permits increasing position by the value transmitted as an argument. These messages make it possible to deviate from a strictly sequential means of proceeding by permitting you to specify targeted read or write positions.

13.2 Sequential Access to Files

This section deals with a few basic operations for processing files. Here we'll restrict ourselves to text files, that is, files that contain exclusively `Characters`.

In order to access the contents of a file, the file must be linked within Smalltalk with a `Stream` object. The streams we dealt with in Sect. 13.1 are also characterised by the attribute *internal*, while those that are linked to files are characterised as *external*. Corresponding to these characterisations, the VisualWorks class hierarchy contains the classes `InternalStream` and `ExternalStream`, which are both subclasses of `PositionableStream` (see Fig. 13.3).

In VisualWorks, files are instances of the class `Filename`, and can be easily created by sending the message `asFileName` to a character string that contains the name of the file. For example, executing the Smalltalk sequence

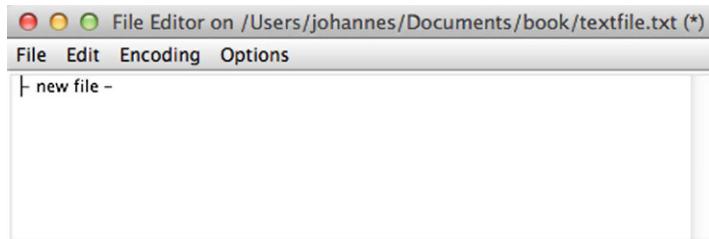


Fig. 13.4 A simple file editor

```
| file |
file := 'textfile.txt' asFilename.
file edit
```

creates a new file called `textfile.txt` in the same directory as the current Image. The message `edit` then opens a window that makes a primitive text editor available. You can use it to manipulate and then save the contents of the file. If this is really a new file, the file displays the contents shown in Fig. 13.4.

If a file with the specified name already exists in the directory, the message `asFileName` opens it, and the editor displays its current contents.

Write Access

When you want to use Smalltalk messages to process the contents of a file, it must first be opened for either read or write access. We'll start out here with write access. For that purpose, the sequence

```
| file stream |
file := 'textfile.txt' asFilename.
stream := file writeStream
```

links the file with an instance of the class `ExternalWriteStream`. If the file already existed, it would now be overwritten. Once it's been created, the `Stream` object can basically be processed with the same messages that were used for the `InternalWriteStream`.

After finishing the “writing work,” the file should be closed by sending the message `close` to the `Stream` object. That ensures that a lock placed on the file by the operating system will be removed, thus making the file available again to other users or programs.

The following Smalltalk sequence writes a character string to the file named `textfile.txt`:

```
| file stream |
file := 'textfield.txt' asFilename.
stream := file writeStream.
stream nextPutAll: 'an arbitrary character string'.
stream close.
Transcript show: file contentsOfEntireFile
```

The message `contentsOfEntireFile` allows the complete contents of the file to be selected.

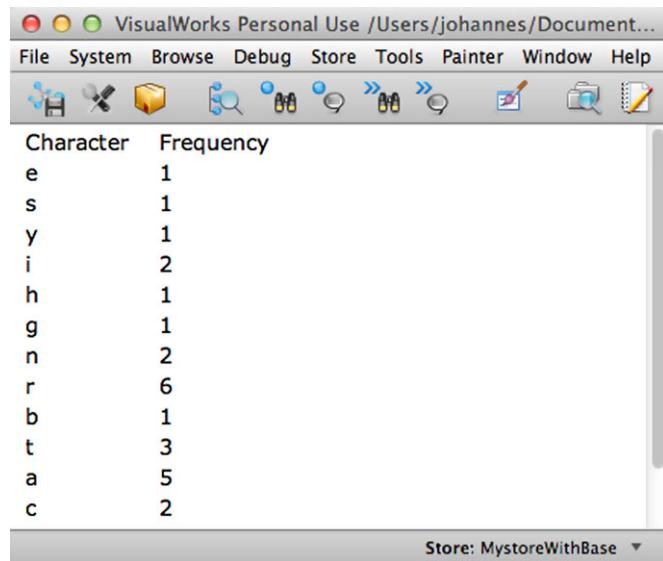
Read Access

For read access to an existing file, the file is bound to an instance of the class `ExternalReadStream`:

```
| file stream |
file := 'textfield.txt' asFilename.
stream := file readStream
```

The following little Smalltalk program reads the text file that was just created and determines how often each letter it contains occurs:

```
| file stream dic |
file := 'textfield.txt' asFilename.
stream := file readStream.
dic := Dictionary new.
[stream atEnd]
  whileFalse:
    [ | character |
      character := stream next.
      character isLetter
        ifTrue:
          [(dic includesKey: character)
            ifTrue: [dic at: character
                      put: (dic at: character) + 1]
            ifFalse: [dic at: character put: 1]]].
stream close.
Transcript clear.
Transcript show: 'character'; tab; show: 'Frequency'; cr.
dic keysAndValuesDo:
```



The screenshot shows a VisualWorks Personal Use window with a menu bar (File, System, Browse, Debug, Store, Tools, Painter, Window, Help) and a toolbar with various icons. A table titled 'Character Frequency' is displayed, listing characters and their counts. A scroll bar is visible on the right side of the table area. At the bottom, a status bar shows 'Store: MystoreWithBase'.

Character	Frequency
e	1
s	1
y	1
i	2
h	1
g	1
n	2
r	6
b	1
t	3
a	5
c	2

Fig. 13.5 Frequency of the letters in ‘an arbitrary character string’

```
[ :character:count |  
 Transcript nextPut: character; tab; tab; tab;  
 nextPutAll: count printString; cr].
```

The result is output to the Transcript (Fig. 13.5).

In conclusion, we should also mention that you can also delete a file by sending the message `delete` to the `FileName` object.

In this chapter, we will examine more closely a few selected basic principles that affect object-oriented programming in general and the structure of Smalltalk programs in particular.

In Smalltalk specifically, conventions for program structures play an important role. Part of this examination also discusses the question of how to structure a class's method protocols. This will be the topic of Sect. 14.1.

Section 14.2 deals with the transformation of various objects into a printable format in the form of character strings. We've already made use of this feature when we used the message `printString`.

A characteristic of object-oriented programs is that the program code is usually divided up into several small methods. In other words, each method solves a clearly delineated small partial task. This doesn't just happen by itself, though; the programmer has to work on it mindfully while designing the program. Section 14.3 discusses some of the basic rules you need to observe to achieve this.

Users today expect to find a graphical user interface (GUI) in a modern interactive application. How to structure such interfaces in general and how to do it with a Smalltalk system like VisualWorks are beyond the scope of this book. Chapter 16 deals with an introduction to the development of Web applications in Smalltalk using the framework system Seaside and shows how to build a Web-GUI.

It's generally true anyway that, when designing an application, it's best to strictly separate the actual application logic—also called business logic—and the way the logic is represented in the GUI. This principle, which is called the *Model-View-Controller (MVC) Paradigm*, was developed along with the development of the GUI for the Smalltalk-80 system; today, though, it has come to be recognised as a fundamental principle for object-oriented development. It is described in Sect. 14.4.

When we use a system of classes to model a segment of the real world, various relationships develop between the classes and their objects. Up until now the only relationship between classes that we have encountered has been the inheritance relationship (see

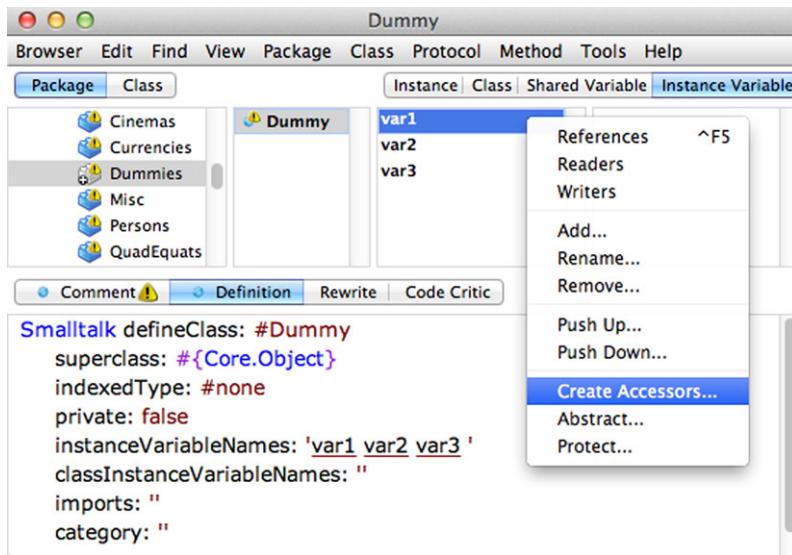


Fig. 14.1 Preparing for the automatic creation of get and set methods

Chap. 6, Sect. 6.1 and Chap. 8, Sect. 8.1.1). In Sect. 14.5, we'll again examine briefly the importance of inheritance for object-oriented modelling of an application. In addition, we'll deal with the relationship types *association* and *aggregation*.

14.1 Standard Method Protocols

In various case studies and in Sect. 11.2 (describing the development of the sample classes Person and Student that we used in that section), we've made a few forays into examining the best way to structure protocols for instance and class methods. We're now going to review this material and expand on it. We'll start out with

Instance Methods

A class that has instance variables usually needs a get and a set method for each of them. The methods are grouped in a protocol called *accessing*. VisualWorks, by the way, has a mechanism for automatically creating get and set methods for a freshly defined class. Figure 14.1 shows the definition of a class called Dummy that has three instance variables. In addition, we've selected the **Instance Variable** tab so that the list of instance variables appears in Field 3 of the System Browser. When the screen looks like this, if we activate the context menu for Field 3 (which also appears in Fig. 14.1), we see the menu item *Create Accessors...*. When we first highlight one or more of the variables and then select that menu item, get and set methods for each highlighted variable are automatically created in the method protocol *accessing*. Figure 14.2 shows that the

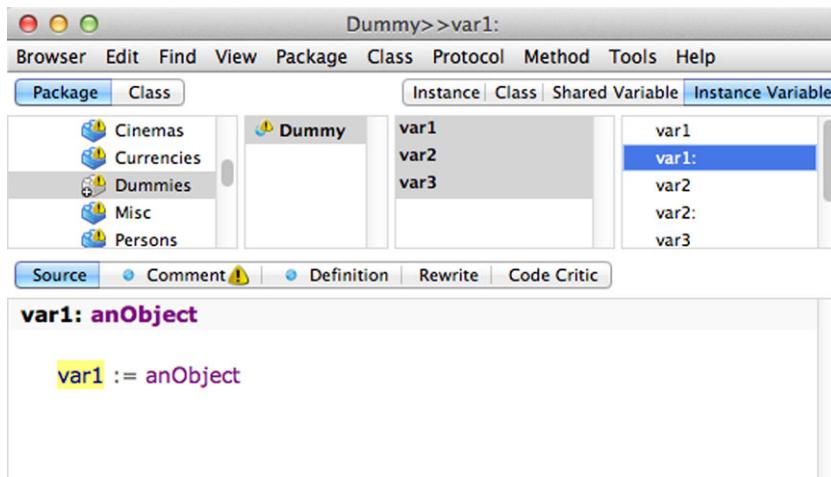


Fig. 14.2 Six automatically created methods

methods are given standard designations: A get method has the same name as the related instance variable; for a set method, which must be a keyword method, a colon is added at the end of the name.

Figure 14.2 also shows that the placeholders required for the set methods are called `anObject` when they are automatically created. It's a Smalltalk convention to give placeholders a name that make it easy to recognise what kind or class of object is expected in their place, for example, `anInteger`, `aPerson`, `anArray`, etc. The set methods would then be similarly named.

In general, we might consider creating get and set methods for all instance variables in the accessing protocol to be a violation of the principle of information hiding. The instance variables can be accessed from outside only by means of these methods, which means that information hiding is formally maintained; in fact, though, the principle is meaningless as long as the get and set methods are part of the public interface of an object. In contrast to various other object-oriented programming languages, Smalltalk does not allow methods to be categorized as not public. Instead, we have another typical Smalltalk convention: All methods that are to be used only within methods of the class itself are stored in the protocol `private`. For that reason, one should reflect for the get and set methods particularly whether they shouldn't also be placed in that protocol, since one might well not want an instance variable to be set from the outside.

Another standard protocol for instance methods is usually called `initializing` or `initialize-release` and uses methods that serve to set the initial values of newly created instances of the class. Figure 14.3 shows the protocol `initializing` for the class `Person`, which in this case contains only a single method. The illustration also shows a protocol `printing` that is always present; it frequently contains only the `printOn:` method. The meaning of which will be explained in detail in Sect. 14.2.

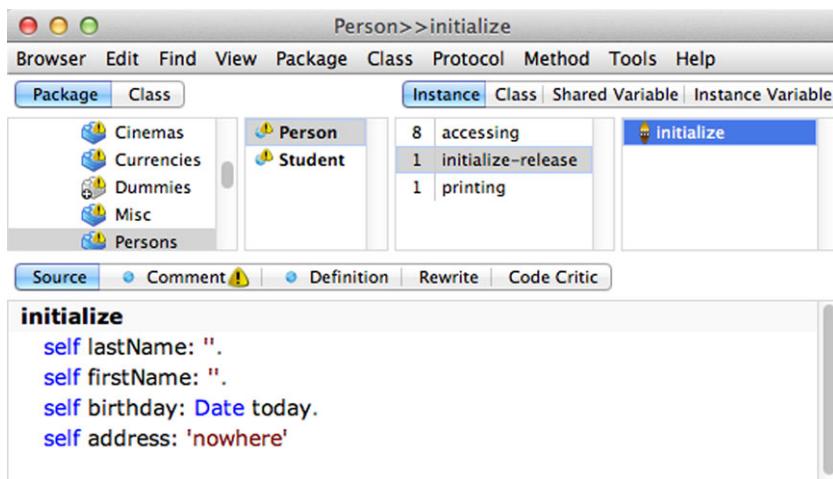


Fig. 14.3 The initializing protocol for the class Person

Table 14.1 Standard composition of a protocol for instance methods

Protocol	Purpose of the Methods	Example
accessing	Get and set methods	
initialising	Setting initial values for newly created instances	initialize initialize: anObject
printing	Transforming objects into character-string representation	printOn: aStream
comparing	Comparing objects	< = hash
converting	Transforming objects into instances of other classes	asOrderedCollection
copying	Creating copies	postCopy
testing	Testing objects for specific characteristics	isEmpty
private	Methods that have meaning only within their classes and should not be used by external access	
...	Application specific	

Additional method protocols are determined and named based on specific applications. At the same time, it's entirely reasonable to base one's names on the names used in the class library to the extent that this seems reasonable. Among these protocols are copying, converting and comparing. Table 14.1 once again provides an overview of the possible structure of a protocol for instance methods.

Table 14.2 Standard composition of a protocol for class methods

Protocol	Purpose of the Method	Examples
instance creation	Creating instances	new new: anObject origin:aPoint1 corner:aPoint2 (See class Rectangle)
examples	Test methods	See Sect. 8.5.6
class initialisation	Initialisation of common variables (class variables)	initialize (See class Date)
private	Methods that have meaning only within their classes and should not be used by external access	
...	Application specific	

Class Methods

Given that the most important task of class methods is the creation of instances of the class, as a general rule, a protocol `instance creation` should be created. Additional protocols usually depend then on specific applications. In Sect. 8.5.6, we noted that it can be useful to define test methods as class methods and to place them in a protocol called `examples`. As far as names for class protocols, the same conventions that apply to instance methods also apply here.

Table 14.2 once again provides an overview of the possible structure of a protocol for class methods.

14.2 The printOn: Framework

The Smalltalk class library contains a built-in mechanism that lets users transform any object into a printable character string (an object of the class `String`). We've already used this in various ways. For example, when we use **Print it** to evaluate an arithmetic expression in Workspace, the result is “printed” in the Workspace window. On this point, you have to know that the window can only display graphics and characters (or character strings).

Another application is to send the message `printString`, which all objects understand. When we want to use the message `show:` to write a number to the Transcript, the number must first be converted into a character string using `printString`, because `show:` only accepts character strings as arguments:

```
Transcript show: (17 sqrt) printString
```

Objects are sometimes converted to character-string representation without our having to do anything, for example, in Inspector and Debugger; we'll return to these situations. The conversion always occurs through activation of an instance method called `printOn:`. Because the class `Object` contains a method by that name, the message is understood by every kind of object.

It's especially practical during the development and testing stages of an application to plan a meaningful text representation for objects of one's own classes. In addition, you must program a dedicated `printOn:` method for the class. Until you do that, inheritance will cause the method defined in `Object` to be activated, which will produce a default text. For example, if for the class `Person` that we introduced in Sect. 11.2 you use **Print it** to evaluate the expression

```
Person new
```

the text

```
a Person.
```

appears. The same thing occurs if you use **Do it** to evaluate the expression

```
Transcript show: Person new printString
```

The default text always consists of the class name preceded by the English indefinite article.

A dedicated `printOn:` method for `Person` objects might look like this:

```
printOn: aStream
  aStream
    nextPutAll:'Person with'; cr; tab;
    nextPutAll:'Last name: '; nextPutAll:lastName;
      cr; tab;
    nextPutAll:'First name: '; nextPutAll:firstName;
      cr; tab;
    nextPutAll:'Address: '; nextPutAll:address;
      cr; tab;
    nextPutAll:'Birthday: '; print:birthday;
      cr; tab
```

The parameter `aStream` must point to an instance of the class `WriteStream` (see Sect. 13.1), which serves more or less as an output medium. We usually use the messages `nextPut:`, `nextPutAll:`, and `print:`, along with `cr` and `tab` for formatting the text (see Sect. 5.3.3).

The expression

```
Transcript show: Person new printString
```

will now display the following text in Transcript:

```
Person with
  Last name:
  First name:
  Address: nowhere
  Birthday: September 24, 2014
```

As a rule, the `printOn:` method is never explicitly called, although it's possible to do so. One could also produce the above text with the expression

```
Person new printOn: Transcript
```

As a rule, the `printOn:` method is activated indirectly, for example when

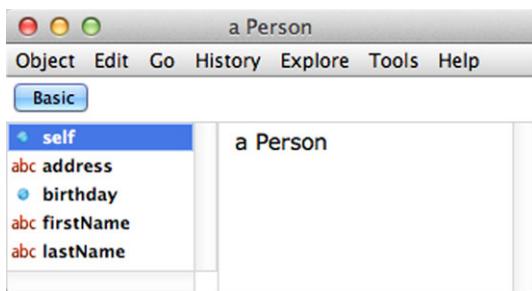
- Using Print it to evaluate an expression
- Sending the message `printString` to an object
- Examining an object in Inspector
- Selecting variables in Debugger

Let's assume for a moment that a `printOn:` method for the class `Person` does not yet exist. In that case, using **Inspect it** to evaluate `Person new` produces the Inspector window shown in Fig. 14.4. If we add the `printOn:` method described above, the image shown in Fig. 14.5 appears.

Now we'll expand the subclass `Student` (see Sect. 11.2) to include a `printOn:` method. We'll again use the method inherited from `Person`:

```
printOn: aStream
  super printOn: aStream.
```

Fig. 14.4 Inspecting a Person object without a printOn: method



```
aStream
cr;
nextPutAll:'Student number: ' print: studentNo; cr;
tab;
nextPutAll:'Major: ';
nextPutAll: major
```

Note, though, that the `printOn:` method in `Person` is programmed so that the text “Person” always appears in the first line. It would certainly be more practical if instead the name of the class of the receiver of the `printOn:` message were to be output. We can accomplish this by making the following change in the `printOn:` method in `Person`:

```
printOn: aStream
aStream
nextPutAll: self class name, ' with'; cr; tab;
nextPutAll:'Last name: ' nextPutAll:lastName; cr;
tab;
```

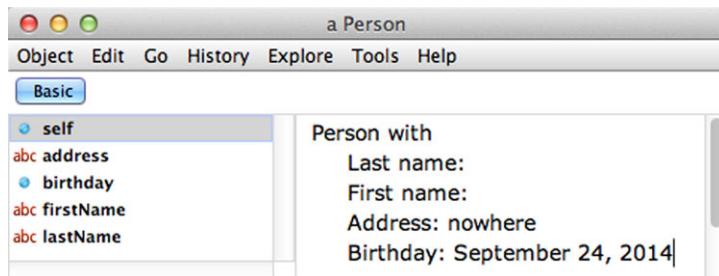


Fig. 14.5 Inspecting a Person object with a `printOn:` method

```
nextPutAll:'First name: '; nextPutAll:firstName; cr;
tab;
nextPutAll:'Address: '; nextPutAll:address; cr; tab;
nextPutAll:'Birthday: '; print:birthday; cr; tab
```

The expression `self class` in the third line delivers the class of the receiver, and the message name, sent to a class, delivers the class name as a character string. Having made this modification, when we use **Print it** to evaluate the expression

```
Student new
```

we now obtain the text:

```
Student with
    Last name:
    First name:
    Address: nowhere
    Birthday: September 24, 2014
    Student number: 0
    Major:
```

It's part of good Smalltalk programming style to define `printOn:` methods early in the development phase of all new classes. The text that these methods produce serve primarily for supporting programmers during testing and quality analysis. That's why it's perfectly practical to output internal structural information about objects that is of no interest to end-users. VisualWorks has a second method available for displaying text for end-users (`displayString` instead of `printString`).

14.3 Transferring Partial Algorithms to Independent Methods

Among the most important results of the intensive technical discussions concerning so-called *structured programming* that occurred during the 1960s and 1970s¹ was the requirement that the readability of the implementation of complex algorithms be achieved, among other ways, by dividing them up into partial algorithms, which would then lead during implementation to a hierarchical system of procedures. Similarly, the rudimentary

¹You can find an interesting description of the topic in Dahl et al. (1972).

method for designing algorithms by *step-by-step refinement* that we described in Sect. 4.1 will lead during implementation—if applied consistently—to a procedure corresponding to an algorithm for solving partial tasks that will invoke additional procedures. Those additional procedures are then in turn able to be hierarchically structured.

This principle, which comes out of procedural programming, has a parallel in object-oriented programming, which requires that programs be assembled from the smallest possible methods. When complex tasks need to be solved, this leads necessarily to the fact that the methods that solve the task must make use of other (auxiliary) methods. In object-oriented programming circles, this principle is called the *Composed Method*.²

To illustrate the principle, let's return once more to the problem of solving a quadratic equation, for which we developed the classes (`QuadrEquat` and `Solution` (with their subclasses) in Sect. 8.5. Based on the solution algorithm written out as a Workspace program at the end of Sect. 2.3.3, we could also have written the method `solveYourself` (see Sect. 8.5.1) in the following way:

```

solveYourself
    "calculates all real solutions for the quadratic
     equation (receiver)"

    | radicand root |
    self solution:
        (a = 0
         ifTrue:
             [b = 0
              ifTrue:
                  [c = 0
                   ifTrue: [TrivialSolution new]
                   iffFalse: [NoSolution new]]
                  ifFalse: [OneSolution with: c negated / b]
              iffFalse:
                  [radicand := b * b - (4 * a * c).
                   radicand = 0
                   ifTrue: [OneSolution with: b negated
                               / (2 * a)]
                   iffFalse:
                       [radicand > 0
                        ifTrue:
                            [root := radicand sqrt.

```

²Beck (1997), for example, discusses it.

```

TwoSolutions
solutionOne:
    (b negated + root)
    / (2 * a)
solutionTwo:
    (b negated - root)
    / (2 * a)]
iffFalse: [NoSolution new]]))

```

This method, which implements the entire solution algorithm, is from the point of view of either hierarchical structuring or the composed-method principle too long. It contains a complex case distinction, which is not easy to gain an overview of. The cases that need to be mathematically distinguished each lead to specific solution procedures. You could also say that a partial algorithm pertains to each case, and that each deals with a specific problem.

The first case distinction deals with the coefficient a . If $a \neq 0$, we're dealing with an “actual” quadratic equation; otherwise it's a linear equation. This consideration then leads to the definition of `solveYourself` that we attempted in Sect. 8.5.1:

```

solveYourself
"calculates all real solutions for the quadratic
equation (receiver)"

self solution:
(a = 0
ifTrue: [self solveLinearEquation]
ifFalse: [self solveQuadraticEquation])

```

This has allowed us to solve two things: For one thing, the trunk of the method has shrunk to three lines and for that reason alone is very easy to comprehend. For another, by introducing new messages and methods, we're now able to give meaningful names to the component partial algorithms. This means that it's not only no longer necessary to add commentaries to the program text, which might have been useful in the original version of the method, it also allows someone reading the program to work directly on the partial algorithms that are of interest at the moment that they're looking at the relevant method.

You can refer to Sect. 8.5.3 to review the definitions of `solveLinearEquation` and `solveQuadraticEquation`. Both methods moreover can only be reasonably invoked from the method `solveYourself`. That makes them both typical examples of the kind of auxiliary methods that we usually store in a `private` method protocol (see Sect. 14.1).

Carving the method `solveYourself` that we presented at the beginning of the section changes nothing in the functionality of the software. Such a change in the program only serves the purpose of making it “prettier”. The kind of process that rewrites a program solely for the purpose of improving its quality with respect to such criteria as its structure, manageability, ease of maintenance, etc. is referred to as *refactoring*.³ Refactoring may not change the functionality of a program, which would, of course, need to be proved. The kind of test methods that we placed in the protocol `examples` in the class `QuadrEquat` (see Sect. 8.5.6) can be very helpful with that kind of proof. That means that, once we have made the changes shown above, calling a testing method of the sort

```
QuadrEquat a: 2 b: 3 c: -4
```

must yield the same result as it did before the changes. That alone is, of course, not proof of the correctness of the modified program, but the converse situation, where the result is not correct, suggests unambiguously that an error occurred during the refactoring. Especially in connection with refactoring you’ll find that regression tests and the use of tools such as the SUnit System can be important (see Chap. 15).

If we were also to subject the method `solveQuadraticEquation` to an intensive study, we might come to the conclusion that it too could be restructured hierarchically. The basic decision point here has to do with the expression beneath the root (`radicand`), which determines whether the quadratic equation has one, two or no real solution. If we modified the method, we might be able to express this in the following way:

```
solveQuadraticEquation
    "calculates all real solutions for the quadratic
     equation (receiver) for the case where a ~=0"

    | radicand |
    radicand := b * b - (4 * a * c).
    ^radicand = 0
        ifTrue: [self calculateTheOneRealSolution]
        ifFalse:
            [radicand > 0
                ifTrue: [self calculateTwoRealSolutions]
                ifFalse: [self noRealSolution]]
```

It’s certainly possible to debate whether it really improves the readability of the method to replace the expression

³On this point see, for example, Fowler (2000) and Sect. 10.4 of this book.

```
NoSolution new
```

in the last line with

```
self noRealSolution
```

The argument for doing so would be that a method should, as far as possible, use messages with the same degree of abstraction. The way we've now defined the method `solveQuadraticEquation`, the reader doesn't need to worry about how the solution objects are created, as long as the only interesting bit is understanding what the algorithm looks like in the case where there's a real quadratic equation. The reader doesn't even need to know about the existence of the class `Solution` and its subclasses. If we also apply this thought process to the method `solveLinearEquation`, for reasons of symmetry, the messages it contains for the subclasses of `Solution` should also be replaced by additional auxiliary methods. At this point though, we won't deal with that.

Now we'll look at the methods we still need:

```
calculateTheOneRealSolution
    "calculates the one real solution for a quadratic
     equation (receiver), expression in the radicand is 0"

    ^OneSolution with: b negated / (2 * a)

calculateTwoRealSolutions
    "calculates all real solutions for a quadratic
     equation (receiver), expression in the radicand is
     positive"

    | root |
    root := (b * b - (4 * a * c)) sqrt.
    ^TwoSolutions
        solutionOne: (b negated + root) / (2 * a)
        solutionTwo: (b negated - root) / (2 * a)

noRealSolution
    "produces the solution object for the case where the
     expression in the radicand is negative."

    ^NoSolution new
```

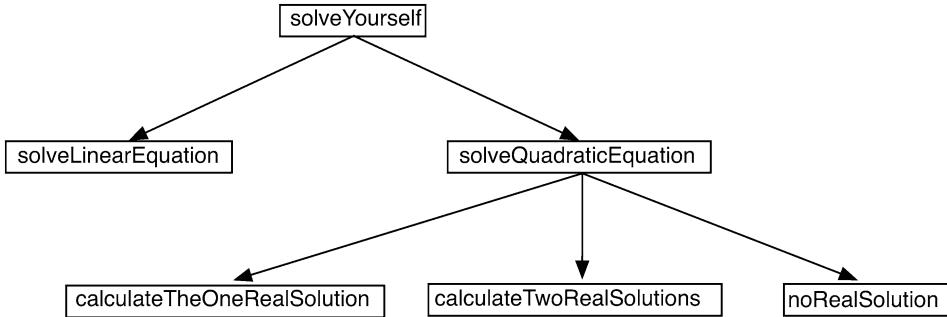


Fig. 14.6 Call hierarchy for the method `solveYourself`

This concludes the hierarchical restructuring of the method `solveYourself`. Figure 14.6 contains a tree diagram of the calling relationships among the methods and the newly introduced auxiliary methods.

We won't hide the fact, though, that dividing up one method into several smaller ones does not provide only advantages. For one thing, activating the divided method `solveYourself` naturally requires the sending of more messages and thus more method activations than the undivided version. Each time a method is activated it costs computer time. But efficiency considerations at this level should not occur until a point where the development of the application is substantially complete. Moreover, it's only in exceptional cases that efficiency problems can be solved by eliminating method calls.

Another problem is that it can be difficult—especially for beginners to object-oriented programming—to keep track of the program run when several smaller methods are used. With greater experience, though, one will increasingly lose the feeling that one needs to probe deeply into the program code of each individual method in order to understand a program. Giving meaningful names to methods helps in being able to abstract from their implementation.

Transferring Partial Algorithms to Independent Methods as a Development Strategy
 Back at the beginning of this section, we mentioned that transferring partial algorithms was related to *step-by-step refinement* in the development of algorithms. When we do that, though, we don't dissect a “finished” algorithm into partial algorithms after the fact; instead we might say that we use this process constructively. As we do that, we develop an algorithm from the top down. In the beginning, we have the first rough description of the algorithm, which essentially describes the division into partial tasks.

With respect to our problem of solving a quadratic equation, we might consider our new method `solveYourself` as a Smalltalk implementation of a high level description of the algorithm. In a second step, one would refine the algorithm by developing the partial algorithms `solveLinearEquation` and `solveQuadraticEquation`.

The difficulty with proceeding this way lies in the fact that, in the process of refinement, one arrives at already existing objects or methods. Object-oriented programming is very

concerned with encouraging the reuse of solutions already in existence. It's possible, for example, that a class for linear equations, together with related solution methods might already exist, and that a top-down way of proceeding might miss its applicability.

For that reason, object-oriented programming likes to use a bottom-up approach, whereby solutions are assembled out of pre-existing partial solutions (modules). If such modules don't already exist for a particular problem statement, then they will be developed before proceeding. In other words, one starts out by developing several small and useful methods. The difficulty with this way of proceeding is that one needs to know in advance which modules are going to be useful. It's possible that certain methods, once they've been developed, will prove to be useless or superfluous.

In practice, one frequently engages in an interplay between top-down and bottom-up processes. At the same time, one should never hesitate to rework a solution that might work, but which is not very attractive in terms of having readable program code; that's the procedure that we demonstrated on the example of the method `solveYourself`.

14.4 User Interfaces—The Model-View-Controller Paradigm

In today's world, the user of an interactive application expects to be able to use it via a Graphical User Interface (GUI). Since the demands made on the ease of use are high, a not inconsiderable portion of the effort that goes into application development goes into designing the GUI.

Furthermore, the user expects that different applications can nevertheless be operated in a similar fashion. That's led to the fact that a number of interface elements such as windows, menus, buttons, input fields, etc. can be found in a variety of places, regardless of the application that's using them.

There are a variety of reasons that argue for separating as much as possible the development of the programming logic that underlies an application from the application's user interface. Here's an example to explain what we mean by this separation. Let's assume that we've been assigned to develop a software realisation of the well-known board game Monopoly. The "business logic" that would underlie a Monopoly application is basically provided by the rules of the game, which describe how the game proceeds, the rights and duties of the players, etc. Independent of the rules, though, we can imagine the following variations of how this application presents itself to the users:

1. The application presents nothing on the screen but an image of the original game. The user has various graphic controls available to be able to play.
2. The purpose of the application is to allow various play strategies to compete against one another. The application starts by establishing the number of players and providing each of them with a particular strategy. From that point on, the games run fully automatically, that is, with no further interactive move from the user. The only result is an announcement of which player won. With this variation, one probably needs only a very simple GUI, or perhaps one can dispense with it entirely.

Regardless of which of the two variations is to be realised, it is still necessary to implement the business logic.

Let's examine the most important reasons for separating business logic and the user interface from one another:

1. The Monopoly example shows that it's entirely possible that one might want to use the same business logic, or portions of it at least, for various applications.
2. Different operating-system platforms usually have specific rules about structuring GUI elements. If you want to create an application to be available on various platforms, you must be able to exchange the GUI. You can do this with a defensible amount of effort only if the business logic and GUI can be easily separated from one another.
3. Certain parts of the behaviour of graphic control elements are independent of the concrete application. For example, when the user places the cursor over a button and clicks the left mouse key, a specific action is supposed to occur. The only application-specific part of this behaviour is the particular action that is to occur. Everything else can be implemented—stated somewhat simply—once and for all in a system of classes and methods for the graphic user controls.

The principle of separating the business logic and GUI from one another began when the Smalltalk-80 development system was developed (Goldberg 1983); it became known as the model-view-controller paradigm. It has proven to be an extremely useful principle for developing interactive applications, and it has come to be widely used within object-oriented development.

Here we will present the basics of the model-view-control paradigm. You can find more extensive descriptions in Goldberg (1983) and Hopkins and Horan (1995).

The three concepts mean the following:

Model means what we've been calling application or business logic.

View means the presentation of the state of the application (the model) with respect to the user.

Controller encompasses the system components that allow the user to interact with the application.

Let's see how these concepts relate to our sample Monopoly application. The classes and objects that would be part of the model implement the rules of the game. A model can have several views. One view might be the representation in a window of the game board with the players sitting around it. Another view that would present the state of the model from a different viewpoint would be the representation of what properties each player owns, with one window for each player. We can think of other points of view relating to the current status of the game.

The controller includes the classes and methods that permit the players to participate in the game; they interpret their interactions with the mouse and keyboard and convert them into messages that are sent to the objects of the model. One player, for example,

selects from a menu the option to acquire a street on which he has just landed. If this move is permitted in the current game status, then it results in a change in the model objects, because the ownership conditions have changed and a payment procedure needs to be initiated. These changes in the model objects might in turn call forth changes in view objects.

As a rule, one controller belongs to each view. Exactly one model is assigned to each view-controller pair, while several view-controller pairs may belong to any one model. In order to be able—as demanded above—to easily separate the application logic (model) from the user interface (view-controller), one must take care that, although the view and controller know their model, they must be informed via a special, automatic-update mechanism about any changes that occur to the model. For that reason, when programming the methods for the model class, it's not necessary to send messages to the view and controller objects.

Changes to model objects are often precipitated by user interactions that the controller converts into messages to model objects. These changes should as far as possible be reflected in all affected views. In order to be able to automate this procedure, the concept of *dependent objects* is introduced. This mechanism allows a list of dependent view and controller objects to be assigned to each model object. All dependent view and controller objects are informed whenever the status of the model object changes, and are thus placed in a position where they can update the onscreen representation of the model object.

The typical interactive cycle between a user and the application therefore looks like this:

1. The user uses the mouse or the keyboard to perform some action (selecting a menu item, clicking a button, etc.).
2. The active controller passes the action along to the model by sending messages to affected objects.
3. Next, the model objects that were addressed undertake to make all necessary changes to themselves to make the state of the application correspond appropriately to the user's wishes.
4. If necessary, the model objects signal to their dependent view and controller objects that their state has changed.
5. The controller objects may also change their behaviour. Once a Monopoly player has selected the option to buy a property, the same option for the same property will thereafter not be offered again to the same player. This change in behaviour is also a result of the change in the model.

Figure 14.7 shows a schema for the flow of messages between the three system components.

VisualWorks offers a framework for implementing the model-view-controller paradigm, which is based on the abstract classes `Model`, `View` and `Controller`.

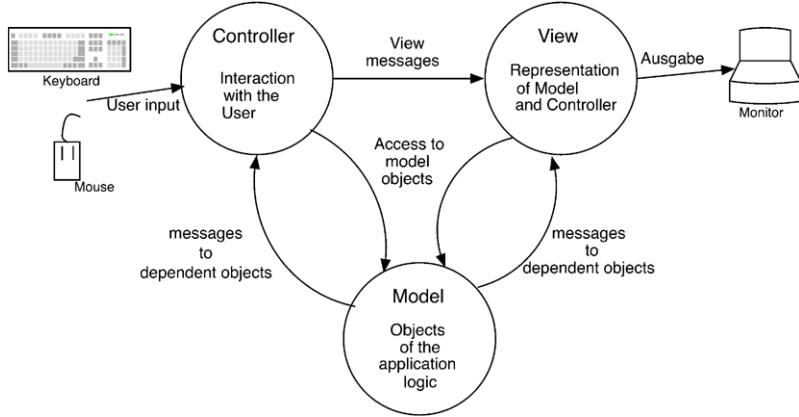


Fig. 14.7 Interaction of model, view and controller

We won't go into greater detail about the procedure for programming GUIs, but we'll simply mention that for classes for application logic, the class `Model` serves as the superclass, rather than the class `Object`. Figure 14.8 shows how the class `Model` fits within the class hierarchy, as well as the comments on the class; it explains that its essential difference from the class `Object` is that it offers the dependency mechanism.

14.5 Relationships Among Classes

In order to present a clear representation of class hierarchies, at various points, we've used so-called class diagrams, the format of which is based on the *Unified Modeling Language* (UML). The UML offers various types of diagrams that help with the documentation of the analysis, design and implementation of object-oriented software without being dependent on any specific programming language. In this section, we'll be making increased use of class diagrams, for which we'll continue to make use of the UML.

Class diagrams are used to show the relationships among classes; up until now, we've limited ourselves to studying the inheritance relationship. As an example, Fig. 14.9 shows the relationship between a superclass and the subclasses that inherit from it.

14.5.1 Inheritance

In Sect. 6.1, we commented that the inheritance relationship between two classes can be interpreted in two different ways. Under the aspect of modelling, a section of the real world through use of a class hierarchy, the relationship “Class A inherits from Class B” is interpreted as “an A object is a B object.”

If we examine Fig. 14.9's class diagram of people at a university, we find that the inheritance relationships in the case mean that:

Model

Browser Edit Find View Package Class Protocol Method Tools Help

Package Class Instance Class Shared Variable Instance Variable

Object Kernel-Objects
Model Tools-IDE-ListIcons

1 copying
2 private

myDependents
myDependents:

Comment Definition Rewrite Code Critic

Any object can have dependents that receive notification of any change to the object. Class Model provides a variant representation for objects that are expected to have dependents; this representation is faster but takes more space than the one provided in class Object.

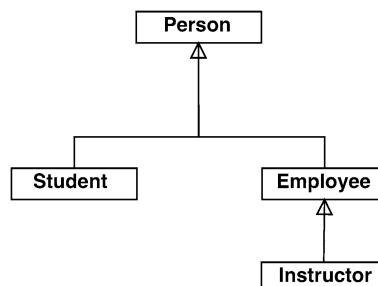
Instance Variables:

```
dependents <nil | Object | DependentsCollection>
```

Object Reference:

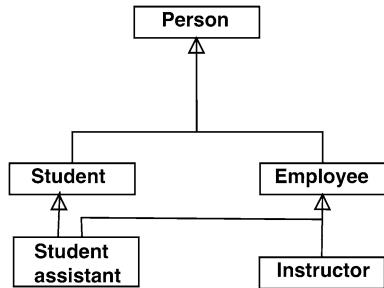
Model is an abstract class whose subclasses represent various kinds of information models. An information model is an object on which user-interface objects such as input fields depend for their data -- thus, the interface objects are said to be dependents of the model. The Model class provides a faster means of finding dependents than the mechanism provided by Object. Here, dependents are kept in an instance variable. By contrast, Object registers dependents for all instances in a single dictionary, which would become quite large if it were routinely used for the dependents of models. For this reason, a new class that is expected to have one or more dependents is frequently made a subclass of Model, when possible.

While Model does not provide any new abilities, it has many subclasses that do. An ApplicationModel mediates between a set of data models and the user interface that is used to manipulate the data. Various kinds of ValueModel are able to adapt simple data objects so they behave like full-fledged models. A SelectionInList adapts a collection of objects for viewing in a list widget, and a SelectionInTable provides a similar service for a table widget's underlying collection.

Fig. 14.8 Meaning of the class Model**Fig. 14.9** Types of people at a university

- A student *is a* person.
- An employee *is a* person.
- An instructor *is an* employee.

Fig. 14.10 An example of multiple inheritance



This kind of relationship between classes is sometimes also called an *isa* relationship.

Two other concepts that are frequently used in this context are *generalisation* and *specialisation*. The class `Person` represents the generalisation of its subclasses; the class `Instructor` is a specialisation of the class `Employee`.

The second way of interpreting an inheritance relationship is of a more technical nature: A class—or its instances—inherits its structure (instance variables) and behaviour (methods) from its superclass. It can sometimes happen that a class hierarchy created under the concept of modelling may prove not be optimal from an inheritance point of view. We'll return to that problem in a little while.

A conceptual hierarchy developed by analysing an application object cannot always be transferred into an isomorphic Smalltalk class hierarchy. For example, if we wanted to expand the university example by the class `Student Assistant`, we could use the argument

- “A student assistant *is a* student.”
- “A student assistant *is however also an* employee.”

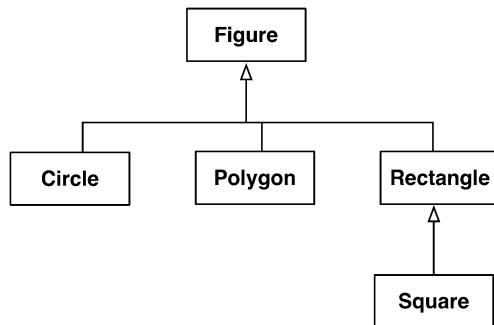
to develop the class hierarchy shown in Fig. 14.10. In this case, the class `Student assistant` inherits from the class `Student` as well as from the class `Employee`. From the viewpoint of application modelling, this is a very reasonable observation. Technically, this is referred to as *multiple inheritance*. Nevertheless, Smalltalk does not permit multiple inheritance; each class in Smalltalk (except `Object`) has precisely one direct superclass.

Some object-oriented programming languages, among them C++ and Eiffel, support multiple inheritance. Smalltalk and other languages that don't allow multiple inheritance make use of some more or less ugly workarounds⁴ to simulate multiple inheritance.

Among the disadvantages of multiple inheritance is the fact that ambiguities arise when two superclasses each have a method with the same name. In that case, it's not clear which method the subclass inherits. Furthermore, the class hierarchy becomes more complex—a tree turns into a directed acyclic graph—and it takes more effort to search for methods.

⁴See, for example, Mössenböck (1994).

Fig. 14.11 Modeling geometric figures



For these and other reasons several object-oriented programming languages decided not to use multiple inheritance.

Reuse vs. Isa Hierarchy

When constructing a class hierarchy that is based on a conceptual hierarchy, technical problems sometimes arise as a result of undesirable structural inheritance. For example, let's look at the modelling of geometrical figures shown in the class diagram in Fig. 14.11. Looked at mathematically, Square is obviously a specialisation of Rectangle. Let's assume that the class Rectangle has two instance variables for the sides *a* and *b*. Structural inheritance though means that each instance of the class Square also has those two instance variables, even though it would be sufficient to supply the length of only one side to describe a square.

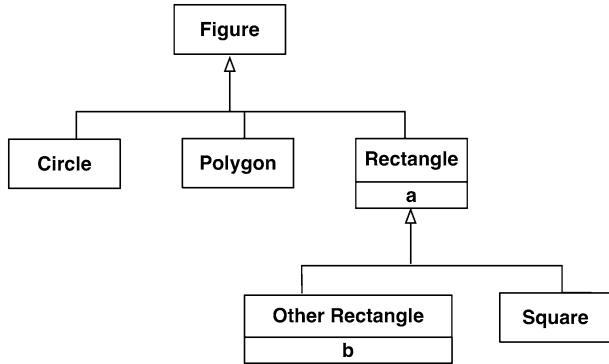
This has two disadvantages. First, it wastes memory when storing squares. Second, you have to ensure that the values of the two instance variables actually are the same for instances of the class Square. That might make it necessary to override the inherited set methods for the lengths of the sides in the class Square.

In order to avoid these disadvantages, we might consider the idea of defining the class Rectangle as a subclass of Square. It would then inherit the length of one side and the pertinent get and set methods from Square, and adds the length of the second side to them. This is a case of optimally using inheritance in the sense of reuse. With regard though to interpreting class hierarchy as application-oriented conceptual hierarchy, it's perverse.

We still need to answer the question: Which of the two options should we use? Above all, when we design a class hierarchy, we should direct ourselves toward an application-oriented conceptual hierarchy. If for technical reasons this should prove to cause problems, we need to check to see if we can accept the related disadvantages. If that's not possible, then we need to think of an alternate class hierarchy; as a rule, though, this may not be as easy as merely inverting the inheritance relationship, as we suggested in the example dealing with rectangles and squares.

The Smalltalk class hierarchy contains a similar problem, which we already mentioned in Sect. 8.1.2. It's related to the number classes Integer and Fraction. From a mathematical point of view, Fraction, as a representative of the rational numbers, would have

Fig. 14.12 Alternative modelling of geometric forms



to be the superclass of `Integer`, which stands for whole numbers; that's because every whole number is a rational number. If that were the class hierarchy though, then the two instance variables for numerator and denominator in the class `Fraction` would be inherited by `Integer`. That would mean that for every whole-numbered object, the denominator, which must be 1, would also need to be stored. The designers of the Smalltalk class hierarchy decided absolutely though that this could not be supported for technical reasons, and thus chose the structure shown in Fig. 8.1, which includes the classes `Fraction` and `Integer` and subclasses of the common abstract superclass `Number`.

If we wanted to apply a similar solution to the rectangle-square problem, we could make the class `Rectangle` an abstract class that has the concrete classes `OtherRectangle` and `Square`, as shown in Fig. 14.12. Instances of the class `OtherRectangle` are all the rectangles that are not squares. For the three classes, the diagram also shows a possible assignment for the instance variables `a` and `b`. In this solution, the problem of wasted memory when storing squares no longer arises. From an application-oriented point of view, the class hierarchy is certainly no longer as elegant as the version shown in Fig. 14.11; nevertheless, it's bearable.

14.5.2 Association

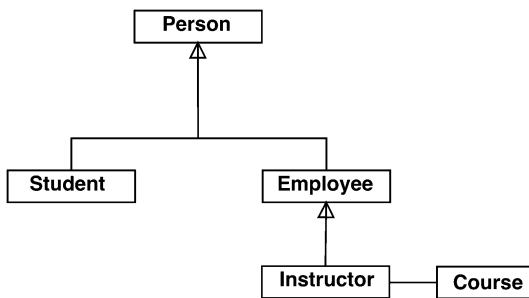
When we speak of an association between two classes, we refer to a relationship in which instances of the two classes have a loose connection with one another. That means that the objects “know” about one another and might therefore cooperate by exchanging messages.

As an example, we'll expand the class diagram for the university from Fig. 14.9 to include an additional class `Course`. Between instances of the classes `Course` and `Instructor` there might be a relationship of this type.

The instructor Einstein teaches the course on the Theory of Relativity.

In the example, the class diagram in Fig. 14.13 uses a connecting line to indicate that these classes have an association relationship with one another.

Fig. 14.13 Association between instructors and courses



While inheritance implies an actual relationship between classes, an association represents a relationship between two objects that are somehow involved with one another. The association relationship in the class diagram indicates that objects of those classes might be involved in such a relationship. Thus the instructor Einstein might teach one, several or no courses.

Association relationships are implemented by instance variables. For example, if one wanted to create an association between a course and an instructor, one could provide an instance variable `instructor` in the class `Course`, which would then contain a pointer to an `Instructor` object. In this case, the cooperation between the associated objects is possible if a course sends a message to an instructor. A `Course` object *knows* its `Instructor` object, but not vice versa.

But it would also be conceivable to define an instance variable `lectures` in the class `Instructor`. It would contain a list (perhaps as an `OrderedCollection`) with pointers to the lectures that are given by the instructor in question.

You can only decide which of the two options should be used in this case by having as a basis an analysis of the operations that are to be performed on the objects of the affected classes. From that, you can determine who should send messages to whom. It's even possible that you would want to allow it mutually for the instances of both classes.

14.5.3 Aggregation

Aggregation is a special type of association. One uses it to model object relationships of the sort

- An A object consists of B objects
- An A object has (possesses) a B object

Aggregation is a containment relationship, where it's also possible to distinguish whether the contained objects are also allowed to exist independently of the aggregate in which they are contained, or whether their existence is dependent on the existence of the aggregate. This second, stricter form of aggregation is also called composition.

Fig. 14.14 Composition relationship between people and their addresses

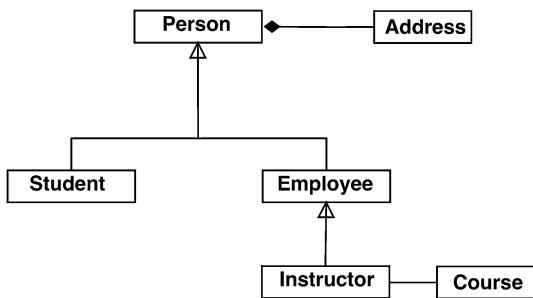
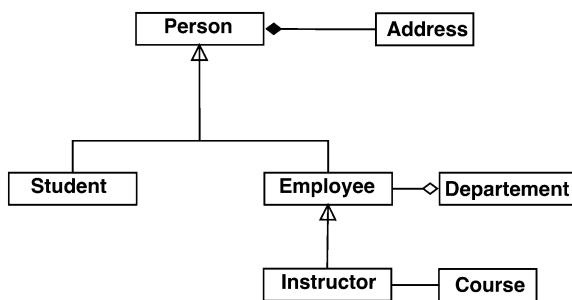


Fig. 14.15 A department consists of employees



An example of a composition is the relationship between a person and their address, where we assume that addresses are instances of their own class. In a class diagram, a composition is represented as a line between the involved classes; the arrowhead of which is a filled-in diamond shape (see Fig. 14.14). Instances of the class **Address** cannot exist independently. When the **Person** object is no longer there, the associated **Address** objects also loses its existence.

As an example of a weak aggregation in which the partial objects exist independently of the aggregate, we could imagine the assignment of employees to certain departments. A department consists of employees who would not be automatically dismissed if the department was dissolved. Figure 14.15 shows a corresponding expansion of the class diagram. In order to represent a simple aggregation, we use a line with an unfilled diamond as the arrowhead.

Everything that has been said about associations also applies to aggregations. Aggregations are regarded as directed associations, so that aggregates recognise their components but not vice versa. That means that the aggregation is implemented through an instance variable in the aggregate class. In the above example of a weak aggregation, one would provide an instance variable **employee** in the class **Department**. Especially with weak aggregations, though, it sometimes happens that the component class also contains an instance variable, so that, in this example, **Employee** methods can send messages to the relevant department.

For compositions as a rule only, the one direction is implemented. It seems strange, for example, for an address to know what person it belongs to.

The character of a composition has as a consequence that, when the containing object is created, the associated instances of the component objects are also created. For example, when a new `Person` instance is created and initialised, an `Address` instance is also created and initialised for that person. It often happens too that particular messages that are received by the aggregate object are forwarded to the component objects. A typical example for that is the `printOn:` method (see Sect. 14.2). If a `Person` object receives a `printOn:` message, it is usually forwarded to the relevant `Address` object.

In today's world, the careful performance of tests is the most important means of quality insurance in the practice of software development. Here, quality assurance means testing to see if the software fulfils the functional requirements expected from it. In this context, a test means the execution of a program and then the comparison of the results of the program performance with the expected results. Strictly speaking, a test is successful when the actual results do *not* agree with the expected results. Since it's safe to assume that complex programs always contain errors, a test is useless when it fails to bring any of these errors to light. The quality of a program can be increased only if an error can be discovered and subsequently corrected. But even in this case a quality improvement is not guaranteed, because a change to a program intended to remove errors can itself insert other errors.

At any rate, testing can only prove the existence of errors. It cannot show whether the software fulfils the demands put on it, because a test always has the character of a random sample.

One alternative to testing is to use formal techniques whose purpose is to prove the correctness of a program. To do this, we use methods we know from mathematics. Since such processes though require a great deal of time and effort, they are now used only when errors in the software might have severe consequences, such as human safety. The topic of software quality is extremely complex, but there's a great deal of literature on the subject, in Ben-Menachem and Marliss (1997) for example, or—particularly on the subject of testing—in Myers et al. (2011).

Tests occur in various phases of software development or at various levels. If software is deemed to be complete, it is tested with acceptance or system tests. In that case, a complete application is tested as a whole. During development, integration tests are performed, which are intended to test the correct interaction of the software's components.

During the development phase, the developer also prepares and performs tests of the individual components. These serve the purpose of checking over individual methods and classes. Such component tests are also sometimes called *unit tests*.

In this chapter, we will be concerned exclusively with the development and automation of component tests.

15.1 Component Tests

In earlier chapters, especially in Sect. 8.5.6, we introduced the technique of writing class methods that could be used to execute portions of the application in order to test them. This kind of program execution, though, is not testing in the strictest sense. That's because a test always includes specifying the expected results; only by doing so can we actually prove whether the actual results of a concrete test run agree with the results that were expected.

It's actually not terribly difficult to expand these "check method" to real test methods; all we need to do is to program into the test method a comparison of the results of the program run with the expected results. Nevertheless, writing this kind of test is an extremely time-consuming task, and so it's practical if the development system supports the programmer in accomplishing this. Moreover, the test methods are only practical when they're actually executed. It takes a great deal of discipline on the part of the developer to regularly activate the test methods. Especially after any program changes, all available test methods should again be run in order to determine whether the program change should now cause any previously successful test to now prove a failure.

► **Note:** Programmers usually say that a test is successful when it yields the expected result. This way of speaking has become common, even though—as we said at the start of the chapter—a test that uncovers no errors is actually useless and thus not successful. In order to avoid any confusion, though, we'll abide by the common point of view, that is, that a test is a failure if it does not yield the desired result.

The systematic execution of all tests after a program change is called a *regression test*. The development environment should also support the programmer in this case. If one or more tests fail, then it's clear that either the changes made to the program added errors to it, or that necessary changes to other parts of the program were not performed.

In the world of Smalltalk programmers, the SUnit System has been available for some time, which is used for automating component tests. This is a system of classes¹ that are also referred to as a framework. It is especially useful for:

- Defining test cases including the expected results.
- Assembling test cases into so-called test suites.
- Automatically running the tests of one or more test suites.

When the tests are automatically run, a display indicates which tests failed, that is, for which tests the expected results did not match the actual results.

¹VisualWorks also contains an SUnit package.

That means that the developer can first report that his portion of the software is complete after all tests have been run without an error.

But as we've already explained, even when a regression test runs without error, that still does not prove that the software is correct. Tests always call forth E.W. Dijkstra's comment, "A test can only prove the presence of errors, never their absence."

Adaptations of SUnit now exist for nearly all programming languages, for example, JUnit for Java.

15.2 Test Automation Using the SUnit

Kent Beck originally developed SUnit (Beck 2013). One can also find current information on the Web (SUnit 2013).

The SUnit framework consist of four classes: `TestCase`, `TestSuite`, `TestResult` and `TestResource`.

- An instance of the class `TestCase` represents a single test case or a group of associated tests.
- An instance of the class `TestSuite` gathers together several instances of `TestCase`, thereby permitting the definition of any number of test cases, which can then be processed together at one specific time.
- An instance of the class `TestResult` assembles the results of a test run. This includes:
 - The number of successful tests
 - The number of failed tests
 - The number of tests aborted because of a program error
- `TestResources` can be used to create an environment (a set of objects) that can be used for running several test cases. This might be used for constructing a database or a network connection.

Using SUnit to develop component tests consists primarily of developing subclasses of the class `TestCase`. In those subclasses, a number of methods are overwritten, which are used to prepare, run and complete a test case for the class to be tested. We'll use a concrete example to see how this is done. For this purpose, we'll return to our test case for the cinema business from Sect. 10.4. First, though, we have to load the package SUnitToo. You can learn how in Appendix A.1.

15.2.1 Test Case: Cinemas

We'll start by writing a test for the method `expensesAt` : in the class `Expenses`, which we implemented in Sect. 7.2.2.

The **first step** consists of defining a test class as a subclass of `TestCase` that includes an instance variable `instance`:

```
Smalltalk.CinemaNs defineClass: #ExpensesTest
  superclass: #{SUnit.TestCase}
  indexedType: #none
  private: false
  instanceVariableNames: 'instance'
  classInstanceVariableNames: ''
  imports: ''
  category: ''
```

It's important that the test class be a subclass of `TestCase` from the namespace `SUnit`. The name of the test class may be chosen at will.

In order to test an instance method for the class `Expenses`, we, of course, need an instance. That's a typical task that needs to be performed to prepare for a test. We take care of this in the test class's `setUp` method.

The **second step** consists of defining the method:

```
ExpensesTest>>setUp
  instance := Expenses new
```

In the **third step**, we program methods that perform the actual tests. That means that they

- Send the message to be tested
- Compare the result with the expected result

To test the method `expensesAt:`, we write the method `testExpensesAt` shown in Fig. 15.1.

Names for test methods must start with the prefix “`test`”. That's the only way to activate them using the so-called *TestRunner* (a component of `SUnit`). The question mark in front of the method name in Field 4 of the System Browser shows that the method `testExpensesAt` is recognised as a test method. The question mark means that the test has not yet been carried out and that the result is therefore not yet known.

The *TestRunner* appears as three buttons in the lower right corner of the status bar of the System Browser.² We will explain later how to use the *TestRunner*.

²If you cannot see the status line, use **Tools → Status Bar** to activate it.

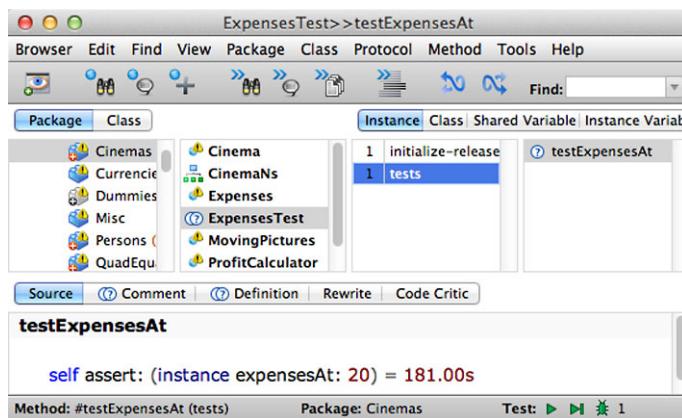


Fig. 15.1 Test for `expensesAt`:

Now let's return to the body of the method `testExpensesAt`. It uses the message `assert:`—inherited from the class `TestCase`—which expects a boolean expression as an argument. That is, the expression must be evaluated to `true` or `false`. If it is `true`, the test is considered to be a success; otherwise it fails. In this specific case, we compare the result of `instance expensesAt: 20` with `181.00s`, that is, the expected value of the expenses with 20 attendees should be 181. The implementation of the class `Expenses` in Sect. 7.2 uses the `initialize` method

```
Expenses>>initialize
self
    perShowing: 180.00s;
    perAttendee: 0.05s
```

to create `Expenses` instances with fixed expenses of 180 and expenses per attendee of 0.05. That means that 20 attendees will yield total expenses of 181.

Successful Test

We can convince ourselves that the test in `testExpensesAt` was successful by clicking the third button from the left (a small green arrow in front of a test tube) to start `TestRunner`. If the test is actually a success, the question mark in front of the method name becomes a green check mark, and the field in the status bar where the three `TestRunner` buttons are located will have a green background.³ `SUnit` marks the successful test with the word `pass`.

³These details on the running of `TestRunner` might, of course, change in subsequent releases of VisualWorks, but they will not affect how `SUnit` works.

Failed Test

If we replace the correct expected value in the method `testExpensesAt` with a different value and then run the test again, the test will fail. The System Browser indicates the result with a red **No** sign in front of the method name and a red bar in the status bar.³ SUnit marks the failed test with the word *failure*.

Aborted Test

A program might not yield the expected result in other circumstances, for example, it might not have run to completion because an exception occurred during the run that led to a program abort. Such exceptions might be caused by an attempt to divide by 0, or by a Message-not-understood situation. In order to bring about such a situation, let's replace the number of attendees in the method `testExpensesAt` with a meaningless character string, for example, '`20`'. The method can still be successfully translated, but when the program attempts to use this "number" to calculate, the program aborts. If we start TestRunner again, the aborted test will be indicated in the System browser with a white exclamation point within a blue circle in front of the method name and a red bar in the status bar. SUnit marks a test that failed to reach the desired result because of these circumstances as an error.

All Three Tests

In addition to the original method `testExpensesAt`, let's now create two additional tests that lead—as described above—to either a *failure* or an *error*:

```
ExpensesTest>>testExpensesAtError
    self assert: (instance expensesAt: '20') = 181.00s

ExpensesTest>>testExpensesAtFailure
    self assert: (instance expensesAt: 20) = 190.00s
```

In order to activate all three tests at the same time in TestRunner, we mark the method protocol but not the individual test methods. After running the tests, TestRunner displays the results window shown in Fig. 15.2. The window title bar shows that

- Three tests ran, of which
- 1 failed and
- 1 caused the program to abort.

The window itself shows the unsuccessful test methods.

Localizing Errors

The simple fact that a test was not successful does not always immediately reveal the cause. You can localise the error that `testExpensesAtError` caused by clicking the

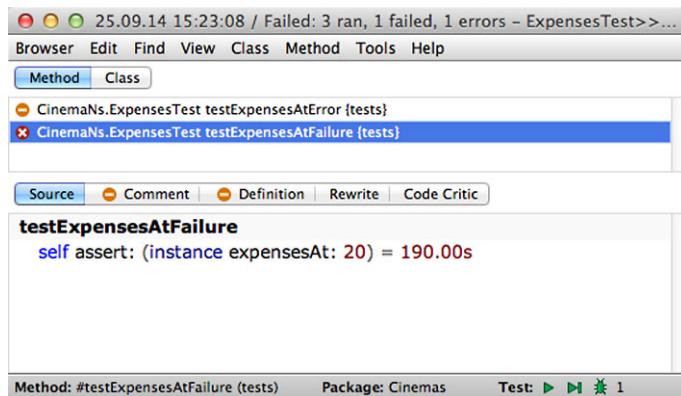


Fig. 15.2 Three tests for `expensesAt`:

second TestRunner button (Debug Tests). The test method then runs in the Debugger, so that one can feel one's way to the location of the error by using the step-by-step procedure, for example.⁴

15.2.2 Additional TestCase Messages

As we explained above, an `assert:` message expects a boolean expression as an argument. If the expression evaluates to `false`, that means that the test failed. The message `deny:` provides the logical complement to `assert::`. If the evaluation of the expression in the argument of `deny:` yields `false`, the test is considered to be a success. That means that the expressions

```
self assert: expression
self deny: expression not
```

are equivalent.

Sometimes we also want to test whether the evaluation of an expression will cause an exception. In Sect. 8.5.7, we examined the possibility of a case where, when solving a quadratic equation, we might try to send the message `solutionTwo` to an instance of the classes `NoSolution` or `OneSolution` so that we could cause a program abort that would yield an error message produced by the message

```
self error: 'Equation does not have two solutions'
```

⁴See Sects. 5.6 and 9.5.

The message `error:` causes an exception that the programmer wants to cause. `TestCase` offers the message `should:raise:` to test if this in fact occurs. For the example shown above, we could formulate the test in this way:

```
self should: [aNoSolutionInstance solutionTwo]
    raise: Error
```

Note that the expression intended to cause the exception must be output as a block after the keyword `should:`. `Error` is the class of the `Exception`. (Exceptions are also objects and thus instances of a class.)

The logical complement to `should:raise:` is `shouldnt:raise::`

15.2.3 An Additional Test for the Class Expenses

The test method `testExpensesAt` that we introduced in Sect. 15.2.1 tests a case where a “standard” `expenses` object is created, which is then initialised using the `initialize` method from the class `Expenses`. Another logical test might consist of creating an `expenses` object using different parameters and then calculating the costs again. To start, it makes sense to create in the class `ExpensesTest` an additional instance variable for the second `Expenses` instance, which we will then place in the `setUp` method. The `setUp` method expanded along those lines might look like this:

```
ExpensesTest>>setUp
instance := Expenses new.
instanceAlternative := (Expenses new)
    perShowing: 250.00s;
    perAttendee: 0.10s
```

We keep the first line in the body of the method so that the test `testExpensesAt` will still run.

For the new instance, we might write the test method:

```
ExpensesTest>>testExpensesAtAlternative
self assert: (instanceAlternative expensesAt:100)
    = 260.00s
```

Once we eliminate the test methods `testExpensesAtError` and `testExpensesAtFailure`, which were only created to demonstrate a point, we are left with two “real” tests for the class `Expenses`, both of which must be successful.

15.3 Test-Driven Development

Responsible programmers test their software extensively, and for economic reasons alone, they should use the capabilities of their development environment while doing so. These days, that includes using test frameworks like SUnit. Making tests replicable in a simple fashion is very advantageous for software quality. Developers who pursue this goal consistently can expect to spend between 25 and 50 per cent of their time working on software writing tests. The degree to which developers supply tests for their classes indicates their faith in their own work. Unfortunately, it's still true that the topic of testing does not receive the attention it deserves in practical software development. Developers frequently work on the principle, "We'll test this later, after we've finished development. Then we'll have time for it." But in many software projects, that time never comes. You can escape this problem only when you start out treating writing code and writing tests as two inseparably connected activities.

As a Smalltalk programmer, whenever you're tempted to write a program in the Workspace (combined perhaps with a test output in Transcript), you should instead immediately invest your effort in a component test.

You should make a habit of writing a test for every method—perhaps with the exception of get and set methods. For a long time, people have even recommended writing the component tests first, that is, before the methods that they are supposed to test. This procedure is referred to as *test-driven development* or the *test-first approach*. When you apply this development methodology consistently, it promises several advantages:

- Each test represents a form of documentation for the components you are testing.
- The danger of putting tests off indefinitely is avoided, at least for component tests.
- When writing tests, you not only need to consider what the method you're testing is actually supposed to do, but also how it should be called, that is, how its interface should be defined. This can lead to improved interfaces.
- It increases the confidence that developers feel in one another's work.
- The tendency to actually undertake necessary refactorings (see Sects. 10.4 and 14.3) increases, because the risk of inserting undiscovered errors while making program changes decreases.

You won't learn how to perform test-driven development consistently out of text books; only practice will help. Nevertheless, we'll recommend two books to lead you further along into the subject: Link (2002) and Beck (2003).

The conventional way to develop applications that are to be manipulated via a GUI is aimed at developing programs that are intended to run on a fully outfitted workstation computer. For large businesses with many workstations and a large number of different application programs, it presents a substantial amount of effort and expense to install the applications on the workstation computers; especially when there are frequent new releases this can lead to high expenses. It's necessary, moreover, to verify that the system's operating platform is installed and kept current so that the application programs are even able to run on it.

One possible way to reduce these costs can involve partially decentralising data processing. That means that the core of the application program is server based, while the user sits in front of a terminal or a thin client that—compared to a workstation computer—is much less powerful. The end-user performs input and output at this thin client. That means that almost no software needs to be installed on the client machine, which therefore requires practically no effort to administer. The application is not delivered to the workstation computers.

Another variation of this scenario is presented by Web applications. In this case too, the core of the application runs on a central computer, the Web server, while the operations occur in a Web browser. These Web applications have already become increasingly significant thanks to the enormous distribution of broadband Internet access. Here too the great advantage is that the applications can, as a rule, run on every client machine that can run a Web browser. No additional software needs to be installed on the workstation computers.

Web applications also represent an alternative to developing applications for mobile devices like smart phones or tablets. Application sellers are independent of having to sell their applications through the often strictly regulated and expensive app stores of certain distributors.

The development of Web applications is being strengthened through the use of frameworks. Such a framework consists of a set of program components that provide certain basic functionalities that are required for interactive Web applications. Developers add

their own program components to this framework, which the framework then activates via a defined interface. Examples of these basic functionalities include the generation of HTML instructions and dialogue control.

Ruby on Rails ([Rails 2013](#)), for instance, has received a great deal of attention. It was developed by David Heinemeier Hansson on the basis of the Ruby programming language ([Ruby 2013](#)), which is in turn loosely based on Smalltalk.

Various frameworks exist for developing Web applications in Smalltalk. The two leading ones are Seaside ([Seaside 2013](#)) and Aida Web ([AIDAweb 2013](#)). The illustrations in this chapter use Seaside. With this framework, it's possible to elegantly write a coherent piece of program code to express the workflow of a Web application that extends across many Web pages. In Seaside, XHTML is created programmatically using conventional Smalltalk methods by sending messages to objects. And the typical tools, like Debugger, that Smalltalk developers know well can be used seamlessly for the development of Web applications.

Seaside was originally developed in the Smalltalk dialect Squeak; in the meantime, though, it's available for other Smalltalk systems including VisualWorks.¹

You can find detailed descriptions of the basics of Seaside in the books by Ducasse et al. ([2010](#)) and Perscheid et al. ([2008](#)), and the website [seaside.st](#) offers a comprehensive overview.

A basic knowledge of HTML and CSS are assumed for understanding the following sections.²

Sections [16.1](#) and [16.2](#) show how, using the simplest of means which are provided by the Seaside framework, an application with a Web interface can be set up. Section [16.3](#) introduces various Seaside concepts that permit the realisation of complex applications that consist of many Web pages. Section [16.4](#) describes how to tie CSS into a Seaside application.

Section [16.5](#) discusses aspects of how to structure Web applications from the point of view of object-orientation, especially how to reuse Seaside components.

At the end of the chapter, Sect. [16.6](#) provides a summary of other Seaside concepts that cannot be described in detail in this book.

16.1 Case Study: Currency Converter

In this section, we plan to show how we can use Seaside to expand the currency converter that we last looked at in Sect. [7.1](#) so that it can be managed via a Web browser. We'll use the model-view-controller paradigm that was introduced in Sect. [14.4](#) to help develop the

¹It is delivered along with Release 8.0 and can be loaded using the Parcel Manager (see Appendix [A.3](#)).

²Good sources of information are [W3SCHOOLS \(2014\)](#) for HTML/CSS and [Ritter \(2009\)](#) for CSS.

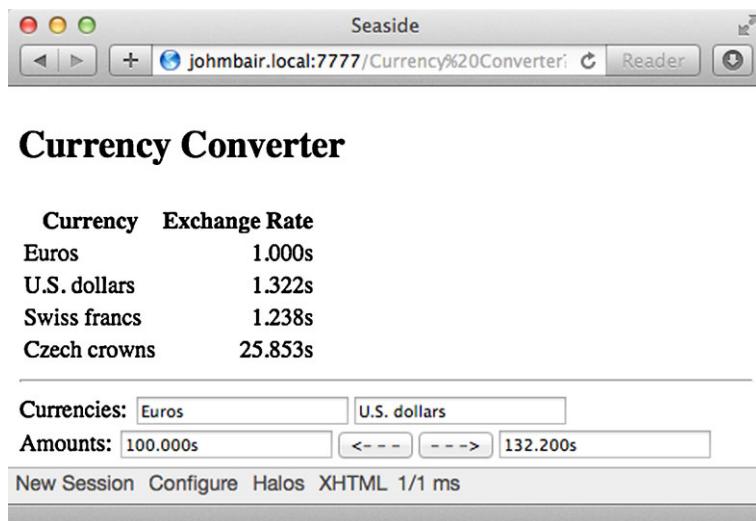


Fig. 16.1 First version of the currency converter displayed in the Web

Web application. That means that we'll entirely sever the logic of the currency converter—which basically consists of a single method that permits conversion of an amount of money from one currency into another—from its representation in a Web browser.

16.1.1 Objective

We plan to develop an initial version of this Web application with the simplest of means, which the user will see looking like the image in Fig. 16.1. Beneath the title, a table will appear showing the currencies stored in the system along with their exchange rates compared to the Euro.

► **Note:** Showing all rates of exchange in relationship to a single base currency is a simplification that is sufficient for our purposes, though it is not entirely compatible with reality.

Beneath the horizontal separator there are four input fields and two buttons available to the user, two each for the currency designations and the corresponding amounts. It's possible therefore to enter two currency designations and one amount and then, by clicking the appropriate button, calculate the amount in the other currency.

Note that, with respect to its layout, this Web page does not satisfy any aesthetic demands, and it leaves a great deal to be desired with respect to user-friendliness. Its design is limited at this point to achieving the desired functionality using the simplest means.

16.1.2 The Model for the Currency Converter

The currency converter in Sect. 7.1 consisted in essence of the class `Converter` with an instance variable `exchangeRate` and the two instance methods

```
Converter>>convert: aNumber
    "converts the amount aNumber at the current exchange
     rate"
    "aNumber * self exchangeRate"
```

and

```
Converter>>convertInverse: aNumber
    "inversely converts the amount aNumber at the current
     exchange rate"
    ^aNumber / self exchangeRate
```

The conversion is thus always based on a single exchange rate. For the Web application though, we want various currencies and their exchange rates to be able to be selected. For that reason, we'll first expand the converter.

Upgrading the Currency Converter

A currency is modelled based on its designation and on its exchange rate with respect to the Euro. That means that an instance of the respective class “knows” how to calculate an amount of Euros into each of the currencies. We can create the class `Currency` most easily by renaming the class `Converter` and adding an instance variable `label`:

```
Smalltalk.CurrenciesNs defineClass: #Currency
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'exchangeRate label'
  classInstanceVariableNames: ''
  imports: ''
  category: ''
```

The two instance methods we used above, `convert:` and `convertInverse:`, remain unchanged. In order to create an instance of `Currency`, we just add the following class method:

```
Currency class>>withExchangeRate: aNumber  
    andLabel: aString  
    "creates an instance of the receiver with the  
     exchange rate aNumber and the label aString"  
  
    ^self new exchangeRate: aNumber; label: aString
```

We would then create the `Currency` instance for the Euro in the following way:

```
Currency withExchangeRate: 1.000s andLabel: 'Euro'
```

► **Note:** Exchange rates and amounts of money are again represented as Fixpoint numbers (see Sect. 7.2) with three places after the decimal.

At this point, we still need a place where all of the instances of the class `Currency`, which our Web application is to make available to the user, can be stored. We are tempted to resurrect the class `Converter` and give it an instance variable `currencies`:

```
Smalltalk.CurrenciesNs defineClass: #Converter  
superclass: #{Core.Object}  
indexedType: #none  
private: false  
instanceVariableNames: 'currencies'  
classInstanceVariableNames: ''  
imports: ''  
category: ''
```

In this instance variable, we will store an `OrderedCollection` with the `Currency` instances. For that, the method

```
Converter>>initialize  
    self currencies: OrderedCollection new
```

takes care of the required initialisation. In order for it to also be activated, it needs its own `new` method:

```
Converter class>>new
  ^super new initialize
```

The actual conversion process should be able to be started with an expression of the sort

```
c amount: 10 from: 'Swiss francs' to: 'U.S. dollars'
```

The variable `c` in the expression points to an instance of the class `Converter`. The method

```
Converter>>amount: aNumber from: aString1 to: aString2
| c1 c2 |
c1 := self currencyWithLabel: aString1.
c2 := self currencyWithLabel: aString2.
^(c2 convert: (c1 convertInverse: aNumber))
```

uses the message `currencyWithLabel:` to fetch the `Currency` object belonging to the label, then calculates the amount first from the source currency (variable `c1`) into the base currency (Euro), and then from the base currency into the target currency (variable `c2`).

And finally, the method

```
Converter>>currencyWithLabel: aString
^self currencies detect: [:c | c label =aString]
```

uses the transmitted label to search in the collection `currencies` for the related currency.

In order to create the screen shown in Fig. 16.1 as simply as possible, we define the following class method. Following our previous conventions, we will store it in the protocol `examples`.

```
Converter class>>converterExample
| c |
c := self new.
(c currencies)
add: (Currency withExchangeRate: 1.0s3
      andLabel: 'Euros');
```

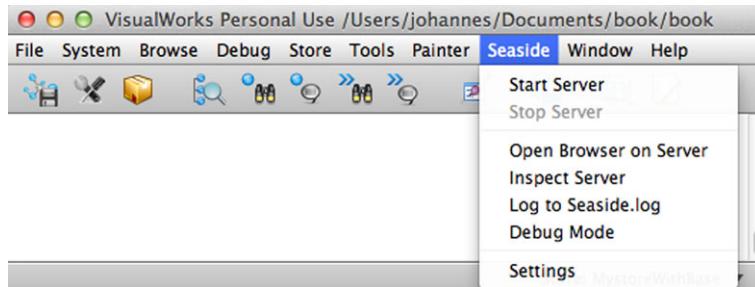


Fig. 16.2 Launcher with Seaside menu

```
add: (Currency withExchangeRate: 1.322s3
      andLabel: 'U.S. dollars');
add: (Currency withExchangeRate: 1.238s3
      andLabel: 'Swiss francs');
add: (Currency withExchangeRate: 25.853s3
      andLabel: 'Czech crowns').
```

^C

At this point, the business logic of the currency converter is complete, and we can turn to the display in the Web browser.

16.1.3 A First Glance at Seaside

Appendix A.3 explains how the packages required for Seaside can be integrated into your own VisualWorks Image. Once you've finished, the launcher will contain a new menu item in the menu bar showing the name Seaside, as shown in Fig. 16.2. Seaside also places a Web server in the Image, which you can start from the menu that is available from this menu item. Once you've started the Web server, select the menu item **Open Browser on Server** to display the page in the Web browser that's shown in Fig. 16.3. There you will find links to

- A few Seaside samples delivered with the framework
- The documentation (The Seaside Book especially is an important source of information) and
- In the right pane, the **Browse** link. After you click this link, you will first land on a configuration page where you can determine whether you always want to see the welcome page when you start the Seaside Web server. Next you come to the **Dispatcher** page, which shows the installed Seaside applications (see Fig. 16.4).

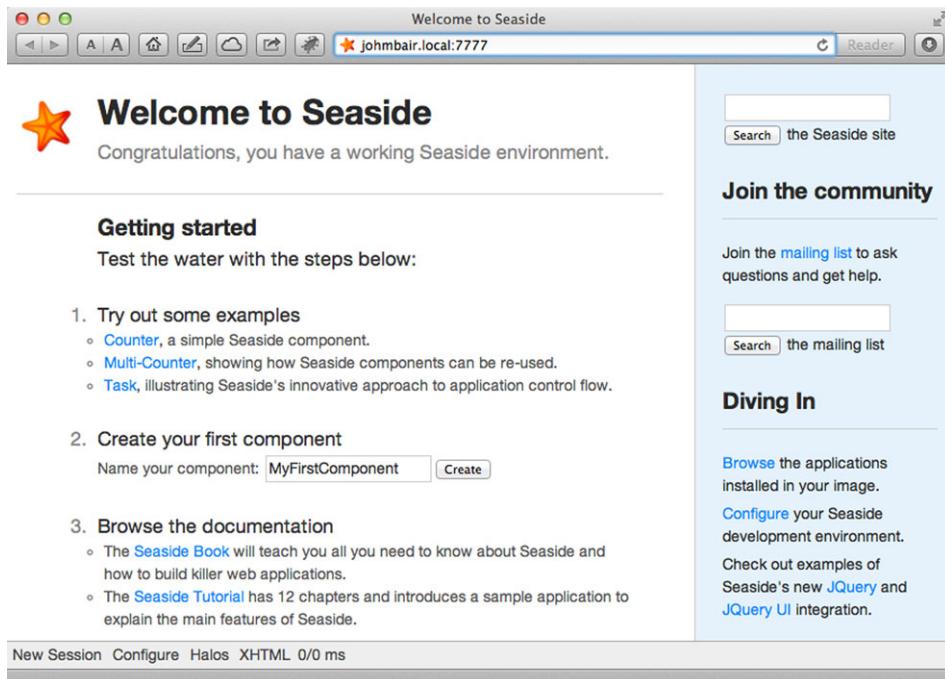


Fig. 16.3 Seaside welcome page

At this point, this page is not very interesting. The link **examples** takes you to additional Seaside examples, while **welcome** returns you to the welcome page shown in Fig. 16.3. Later on, our currency converter application will appear on this page.

► **Note on the address line in the Web browser:** The address consists of the local computer name followed by a colon and then the port number 7777. You can also change the port, though, using the menu item on the Launcher **Seaside → Settings**. In place of the actual computer name, you can also use **localhost**.

16.1.4 Realising the Web Interface

The Seaside framework is completely programmed in Smalltalk. All of its classes and methods are available from the Image. It is used by sending messages to Seaside objects and by preparing one's own methods.

These methods are activated using messages from the framework. Figure 16.5 illustrates this relationship.

The most important messages that are sent to Seaside are the ones that create HTML code. However, one no longer “programs” in HTML, but rather exclusively in Smalltalk.

Fig. 16.4 The applications installed in Seaside

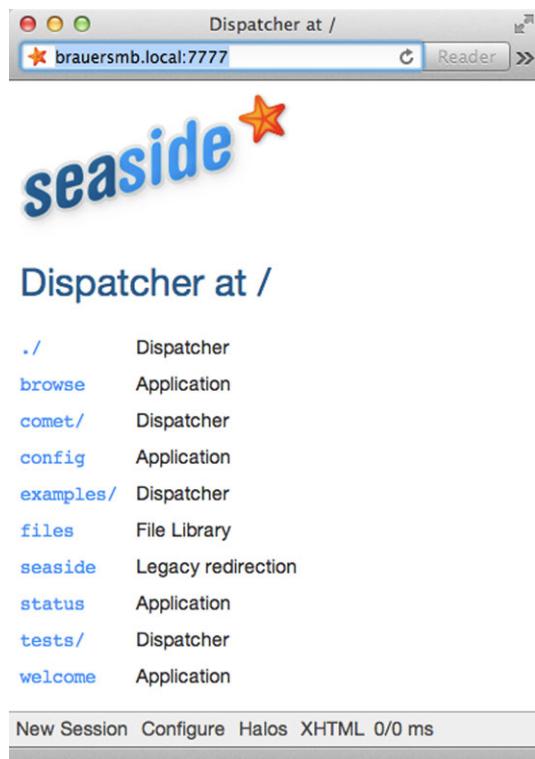
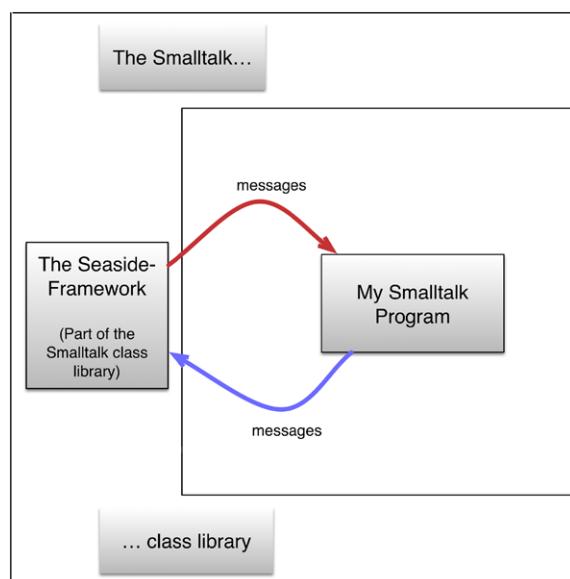


Fig. 16.5 Embedding a Web application in the framework



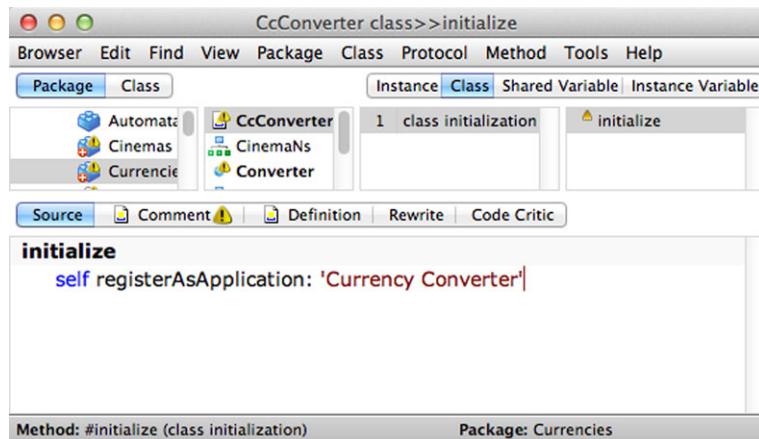


Fig. 16.6 The class method `initialize`

In Seaside, each browser page is implemented by one Seaside component. It happens, though, that such a component is nothing more than an instance of a subclass of `Seaside.WAComponent`. The class for our Web application is called `CcConverter` and is defined as follows:

```
Smalltalk.CurrenciesNs defineClass: #CcConverter
  superclass: #{Seaside.WAComponent}
  indexedType: #none
  private: false
  instanceVariableNames: 'converter currency1 currency2
                        amount1 amount2'
  classInstanceVariableNames: ''
  imports: ''
  category: ''
```

As always, the class name can be anything you choose. We're using the prefix `Cc` here to indicate the classes whose instances are Seaside components. Five instance variables are defined in the class definition; we'll explain their purpose later.

Registering a Class as a Seaside Application

In order for our currency converter to appear on the Seaside Dispatcher page (see Fig. 16.4), a Seaside component must have a class method `initialize`. Figure 16.6 shows an appropriate method for the currency converter in the System Browser. The message `registerAsApplication:`³ places the name of the link on the Dispatcher page.

³The relevant method is inherited from the class `WAComponent`.

Fig. 16.7 The currency converter has been installed



In order for the registration to actually occur, the message must be sent. The first time this has to be done manually. In the System Browser, highlight the body of the method (without `initialize`) and perform it once with `Do it`. Afterwards, the new application appears on the Dispatcher page. (See Fig. 16.7. It may be necessary to reload the page in the Web browser.) As long as the class `CcConverter` contains no methods other than the class method described above, an empty page will appear in the Web browser when you click the `currency converter` link.

The `renderContentOn:` Method

The following steps occur when you click a link—such as `currency converter`—on the Dispatcher page:

1. The Seaside server receives an HTTP request, which means that the Web browser uses the HTTP protocol to send a request to the server. Essentially, this means that the contents of the address line of the Web browser are transmitted.
2. The server checks whether a Seaside application is registered for the link name; in our case, executing the class method `initialize` (see Fig. 16.6) registered the application.

3. Seaside creates an instance of the related component class (`CcConverter` in this case) and sends it two messages:
 - a. The message `initialize`:—In a corresponding instance method, the component can perform the initialisation of its instance variables. We'll deal with that later. If the component does not have an `initialize` method,⁴ a method inherited from the superclass `WAComponent` will be activated.
 - b. The message `renderContentOn`:—This allows the component to generate HTML code, which is then rendered on the Web page belonging to the component. Until the `renderContentOn`: method exists, the Seaside server responds to the HTTP request with an empty Web page.

A simple `renderContentOn`: method, which would do nothing but create the title shown in Fig. 16.1, might look like this:

```
CcConverter>>renderContentOn: html  
    html heading: 'Currency Converter'
```

When a `renderContentOn`: message is sent, Seaside uses the `html` parameter to transmit an “HTML drawing board” to the component, on which HTML elements can be “drawn” by sending messages to the `html` object. The `heading:` message creates an `<h1>` element.

► **Note:** The `renderContentOn`: method works a little like the `printOn`: method described in Sect. 14.2. At that point, we were already speaking of a framework. As a rule, these methods are not activated by messages from their own application. The activation always occurs from the environment. Within its own class, the corresponding method just needs to be waiting. In the case of Seaside, the environment is the Seaside Web server.

With the `renderContentOn`: method described above, our Web page now looks like Fig. 16.8.

At this point, we will cast an initial glance at the use of the developer tools that Seaside makes available on the Web page. On the lower edge of the screen, you can see the Seaside status bar. Among other things, this has a link called **Halos**. If we click this link, the appearance of the Web page changes as shown in Fig. 16.9:

- The portion of the Web page that was created by the component `CcConverter` is enclosed within a frame.
- Three symbols appear beside the class name (described here from left to right):

⁴Note that we're dealing with the *instance* method here, not the *class* method.



Fig. 16.8 The title



Fig. 16.9 The Seaside Halos

- The first opens the System Browser for the class.
- The second starts an Inspector to run within the Web browser.
- The third allows input of CSS commands. We will not make use of this option, however.
- The links **Render** and **Source** appear on the right edge of the title bar of the frame. **Render** is usually selected.

If you select **Source** instead, the Seaside server creates the Web page shown in Fig. 16.10. It shows the `<h1>` element, which is the code that the `CcConverter` component used to create the HTML fragment. Note that this is naturally not the complete HTML code for the page. After all, the code also has to be used to create the Halos, status bar, etc. But developers are usually interested only in the HTML fragment that they programmed.

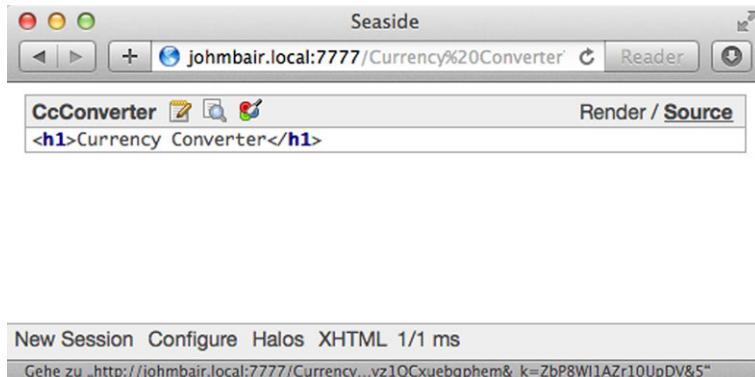


Fig. 16.10 The created HTML code

Designing the Currency Table

Next, we want to expand the `renderContentOn:` method so that the table with the four currencies appears beneath the title, as shown in Fig. 16.1. We need to resolve two questions before we can do that:

1. How can the `CcConverter` component access the currencies stored in an instance of the class `Converter`?
2. Which messages are used to create HTML tables?

Let's try first to answer the first question. At the end of Sect. 16.1.2, we defined the class method `converterExample` in the class `Converter`, which creates an instance of this class with the four currency examples. The Seaside component `CcConverter` must be able to access such an instance in order to be able to display the currencies as a table. Because the component will need to send various messages to the `Converter` instance, the instance must be stored in an instance variable. For this purpose—as we already described in Sect. 16.1.4—we prepared for this eventuality by providing the class `CcConverter` with the instance variable `converter`. Now we can use this variable in the method

```
CcConverter>>initialize
super initialize
self converter: Converter converterExample
```

Now the Seaside component “knows” its model. In this case, it's not necessary to activate this method by defining a class-specific new method with a corresponding `initialize` message. When Seaside creates an instance of a subclass of `WAComponent`, it automatically sends an `initialize` message. In the `initialize` method you create, though, it's absolutely essential to use the expression `super initialize` to activate



Fig. 16.11 The column headings

the method of that name in the superclass `WAComponent` so that the inherited instance variables⁵ can be initialised correctly.

Now let's turn to the second question. So that we can first of all create the column headings in the table, we will extend the `renderContentOn:` method in the following manner:

```
CcConverter>>renderContentOn: html
html heading: 'Currency Converter'.
html table:
[html TableRow:
[html
tableHeading: 'Currency';
tableHeading: 'Exchange Rate']]
```

The message `table:` creates an HTML table element; the contents of the table must be transmitted in a block as an argument. At this point, the content consists only of a single row, which was created using the message `TableRow:`; its contents must also be enclosed within a block. Column headings are created using the message `tableHeading:`. Figure 16.11 shows the result. Figure 16.12 shows the HTML code created by the Seaside messages.

We must now create a row in the table for each of the four currencies. We can use the `get` method `currencies` to access the currencies, which yields an `OrderedCollection` with instances of the class `Currency`. We iterate with the `do:` message over this collection so that a row is created for each currency showing in one column its label and in the other its exchange rate. The `renderContentOn:` method now looks like this:

⁵In this case, specifically the variable decoration.

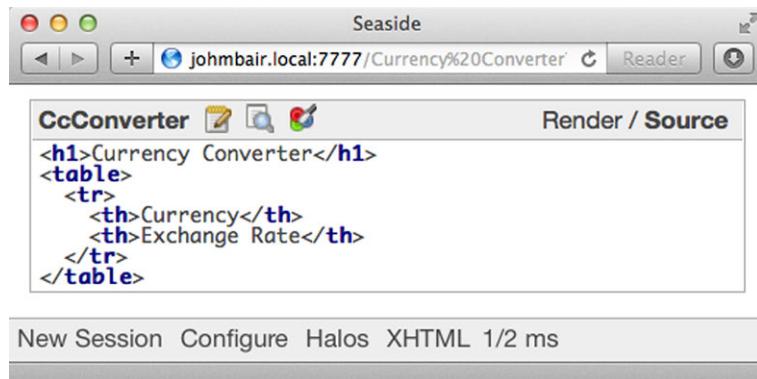


Fig. 16.12 The HTML code for the column headings

```
CcConverter>>renderContentOn: html
  html heading: 'Currency Converter'.
  html table:
    [html TableRow:
      [html
        tableHeading: 'Currency';
        tableHeading: 'Exchange Rate'].
      self converter currencies do:
        [:c |
        html TableRow:
          [html
            tableData: c label;
            tableData: c exchangeRate]]]
```

The result is shown in Fig. 16.13. Nevertheless, the image doesn't agree yet with the one in Fig. 16.1; the exchange rates in this table are left-aligned rather than right-aligned.

We can right-align the exchange rates in the table by replacing the expression

```
html TableRow:
  [html
    tableData: c label;
    tableData: c exchangeRate]
```

with the following:



Fig. 16.13 Table with four currencies

```
html TableRow:  
  [html tableData: c label.  
   html tableData align: 'right'; with: c exchangeRate]
```

Once we've made that change, the table on the Web page will look like the one in Fig. 16.1. This was accomplished by replacing the message `tableData:` with `tableData`. The message `align: 'right'` then supplies the `<td>` element created by the change with the corresponding HTML attribute `align="right"`. The cascaded message `with: c exchangeRate` then establishes the content of the `<td>` element.

► **Notes on the structure of Seaside messages:**

1. If you simply want to define content for an HTML element, as we did above with the currency label, you can use the keyword message assigned to the HTML element—in this case, `tableData:`—and send the content directly as an argument. If you want to supply HTML attributes to an HTML element though, you must instead use the corresponding unary message `tableData` here. You can then assign in the form of a message cascade any attributes to the HTML you are creating. The final message in the cascade is then `with:,` to which you supply the content of the element as an argument. You can also use this format though when you do not want to supply any attributes. That means you can look at the keyword message `tableData: c label` as a short version of `tableData with: c label`.
2. If the content of an HTML element consists solely of a simple value (a character string), the content can be supplied as an argument of the corresponding keyword message—for example, `tableData:—`or of the `with:` message. Furthermore, when you're

dealing with numbers it's not necessary to use `printString` to transform them into character strings; Seaside performs the conversion automatically. If, however, an HTML element consists of subelements—for example, `<tr>` elements created using `tableRow:`—then the subelements must be enclosed within a block.

► **Note on formatting table contents:** The right alignment of the exchange rates that we accomplished using the HTML attribute `align` is in fact a workaround. When dealing with amounts of money, it's usually preferable to align rows at the decimal point. Most Web browsers, though, don't currently translate `align="char"`, the HTML attribute used to perform this alignment. On the other hand, though, such formatting questions are more properly a topic for CSS. Yet CSS also has no elegant solution to the problem.

Displaying Input Fields and Buttons

Beneath the horizontal separator (see Fig. 16.1) you can see two labels, four input fields and two buttons. The next section explains how they are created.

First, though, we want to refactor the `renderContentOn:` method. The example we created earlier already shows how a large number of HTML elements on a Web page can quickly make `renderContentOn:` methods large and confusing. Our Web page actually consists of three parts:

1. The title
2. The currency table
3. The form with the input fields and buttons

Based on that analysis, the `renderContentOn:` method should consist of three expressions:

```
CcConverter>>renderContentOn: html
    html heading: 'Currency Converter'.
    self renderExchangeRateTableOn: html.
    self renderFormOn: html
```

That means that we'll introduce two auxiliary methods. In the method

```
CcConverter>>renderExchangeRateTableOn: html
    html table:
        [html TableRow:
            [html
                tableHeading: 'Currency';
                tableHeading: 'Exchange Rate']].
```

```
self converter currencies do:  
    [:c |  
        html TableRow:  
            [html tableData: c label.  
             (html tableData)  
              align: 'right';  
              with: c exchangeRate]]]
```

we export the creation of the currency table, and in the method `renderFormOn:`, we program the rest of the Web page:

```
CcConverter>>renderFormOn: html  
    html  
    horizontalRule;  
    form:  
        [html  
         div:  
             [html text: 'Currencies: '.  
              html textInput.  
              html textInput];  
         div:  
             [html text: 'Amounts: '.  
              html textInput.  
              html submitButton: '<---'.  
              html submitButton: '--->'.  
              html textInput]]
```

First a horizontal line is created, followed by an HTML form created using the message `form::`. In HTML, input fields and buttons must always be enclosed within a `<form>` element. The form in turn consists of two HTML `<div>` boxes that were created using the message `div::`. There are two reasons for this: first, the boxes are written in two rows in the Web browser and second, this format makes it easier at some point in the future to use CSS commands to format the form area of the screen.

The upper `<div>` box contains simple text and two text-input fields that were created using the message `textInput`. In the lower `<div>` box, two input fields surround two buttons, which were generated using the message `submitButton::`. The labels on the buttons are passed as character-string arguments.

The result in the Web browser now matches the graphic in Fig. 16.1 pretty exactly, except that the input fields are empty. It's possible to place certain default values in those fields. In order to do that, you must send the message `with:` to a `textInput` object, using the value you want displayed as an argument. We can add to the `renderFormOn:` method in the following way:

```
1 CcConverter>>renderFormOn: html
2   html
3     horizontalRule;
4     form:
5       [html
6         div:
7           [html text: 'Currencies: '.
8             html textInput with: self currency1.
9             html textInput with: self currency2];
10        div:
11          [html text: 'Amounts: '.
12            html textInput with: self amount1.
13            html submitButton: '<---'.
14            html submitButton: '--->'.
15            html textInput with: self amount2]]
```

We've changed lines 8, 9, 12 and 15. For values, the contents of the instance variables `currency1`, `currency2`, `amount1` and `amount2` are used. We already defined these instance variables when we introduced the class `CcConverter`. In order to give them meaningful values, we could also expand the `initialize` method as follows:

```
1 CcConverter>>initialize
2   super initialize.
3   self converter: Converter converterExample.
4   self
5     currency1: 'Euros';
6     currency2: 'Euros';
7     amount1: 0.0s3;
8     amount2: 0.0s3
```

Lines 4 to 8 were added. Figure 16.14 shows the result after these various changes were made. Now the desired interface has been constructed but it is still mostly useless. Although the user can enter amounts in the input fields, clicking the buttons only causes the default values to continue to appear. The main reason for that is that no one has defined

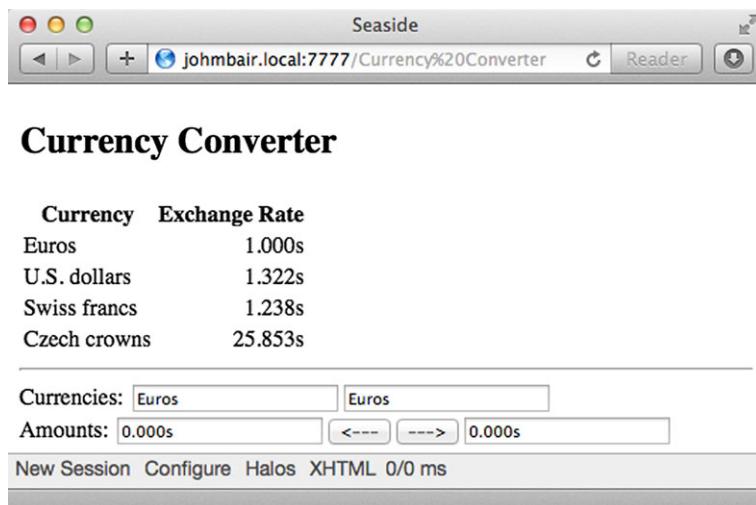


Fig. 16.14 Input fields with default values

what is supposed to happen when you click on one of the buttons. That's the subject of the next section.

16.1.5 Implementing the Functionality

At this point, we have to first make a couple of comments concerning communication between the Web server and the Web browser. A Seaside application runs only on the Web server. When a user does something on a Web page, the Web browser is active. In order to activate some function on the Web server, the Web browser has to send an appropriate request, called an HTTP request, to the Web server. That might happen when the user clicks a button. When such an HTTP request is sent, then the content of all the fields of the related form is sent to the Web server. But no HTTP request is sent if the content of an input field is merely changed without clicking a button.

Seaside's Callback Mechanism

For each element of a form (that is, of a `<form>` element), Seaside allows you to specify what will happen on the Web server when the Web browser sends an HTTP request.

In Seaside, the specification of such actions occurs through use of the message `callback:` or through its argument sent as a block. For the first currency field, that might look like this in the `renderFormOn:` method:

```
...
(html textField)
```

```
callback: [:input | self currency1: input];
with: self currency1.

...
```

Seaside evaluates the block once an HTTP request occurs. The character string that the user enters in the input field is assigned to the block parameter (`input` in this case). In the body of the block, this character string is then assigned to the instance variable `currency1`. As before, the `with:` message defines the value that is to be displayed in the Web browser.

Once you have performed this modification to the `renderFormOn:` method and entered, say, the text “Swiss francs” in the first input field, that value will remain in place after you click one of the buttons. This behaviour occurs because Seaside again sends the message `renderContentOn:` to the component after all callbacks after an HTTP request have been processed. (This time around, Seaside does not send first an `initialize` message.)

Now we’ll make a similar expansion to the second currency-input field. We do things a little differently for each of the two input fields. The content of an input field is always a character string. However, `FixedPoint` numbers are stored in the instance variables `amount1` and `amount2`. Therefore, the character strings must first be transformed into `FixedPoint` numbers before we can proceed. The corresponding `callback:` message looks like this:

```
...
(htmltextInput)
callback: [:input | self amount1:
            (input asNumber asFixedPoint: 3)];
with self amount1.
...
```

The message `asNumber` first changes the character string in `input` into a number; then the message `asFixedPoint: 3` changes it into an instance of `FixedPoint` with three places after the decimal. The `callback:` message for the second amount field looks the same.

All that’s missing now is the initialisation of the actual conversion of the amounts that occur when we click one of the two buttons. If the button labelled ‘`>`’ is clicked, the amount in the left field is to be converted from the currency indicated in the left currency field into the currency indicated in the right currency field; the result will be displayed in the right amount field. When we click the button labelled ‘`<`’, the same things occurs, but working from right to left. In order for that to happen, the two submit buttons must be supplied with appropriate callbacks.

The complete renderFormOn: method now looks like this:

```
1 CcConverter>>renderFormOn: html
2   html
3     horizontalRule:
4     form:
5       [html
6         div:
7           [html text: 'Currencies: '.
8             (html TextInput)
9               callback: [:input | self currency1: input];
10              with: self currency1.
11            (html TextInput)
12              callback: [:input | self currency2: input];
13              with: self currency];
14          div:
15            [html text: 'Amounts: '.
16              (html TextInput)
17                callback:
18                  [:input | self amount1:
19                    (input asNumber
20                      asFixedPoint: 3)];
21                  with: self amount1.
22                (html submitButton)
23                  callback:
24                    [self amount1: (self converter
25                                  amount: self amount2
26                                  from: self currency2
27                                  to: self currency1)];
28                  with: '---'.
29                (html submitButton)
30                  callback:
31                    [self amount2: (self converter
32                                  amount: self amount1
33                                  from: self currency1
34                                  to: self currency2)];
35                  with: '-->'.
36                (html TextInput)
37                  callback:
38                    [:input | self amount2:
39                      (input asNumber
40                        asFixedPoint: 3)];
41                  with: self amount2]]
```

The changes occur in lines 22 to 35. The class Converter provides the instance method amount:from:to: to perform the currency conversion. In the callback for the first button ('---'), this method is activated by the message

```
(self converter amount: self amount2  
    from: self currency2  
    to: self currency1)
```

which converts the right currency amount (`amount2`) from its currency (`currency2`) into the target currency (`currency1`). The result is then stored in the instance variable `amount1`.

For the second button ('--->'), everything happens the same way, but in reverse.

This completes the first version of the currency converter. Nevertheless, it still has a few serious shortcomings.

16.1.6 Shortcomings in the First Version

Our first Seaside application allows the conversion of only a limited number of currencies, and then only when the user doesn't make any errors when inputting values. We should bear the following weak points in mind:

1. Inputting currency labels in the text fields is not just uncomfortable but also subject to mistakes.
2. The input of the currency amounts also presents problems: If we enter a character string in a text field that can for some reason not be converted into a number, both text fields will display the number 0.000s.
3. The list of currencies is hard coded into the program. Changing the list means changing the program.
4. The same thing is true for the rates of exchange.

The first two points deal with the usability of the application from the point of view of the end-user, who only wants to convert currencies. Section 16.2 deals with improving usability.

The last two points are directed more toward the application's administrator, who is responsible for maintaining the currencies and their rates of exchange. Section 16.3 introduces an administrative interface.

Finally, Sect. 16.4 looks at the question of layout design. It deals especially with the question of how to integrate CSS into a Seaside application.

16.1.7 Refactoring the Method `renderFormOn`:

The next steps in the development will deal primarily with further development of the instance method `renderFormOn`: from the class `CcConverter`. Since that method

has already become quite long though, we'll start out by introducing tools to divide it up hierarchically. In order to do this, we'll make the following changes:

- The steps for rendering the two currency input fields will be exported to their own method, `renderCurrencyFieldsOn:`.
- We'll also introduce specific methods for rendering:
 - The left amount field (`renderAmountField1On:`)
 - The right amount field (`renderAmountField2On:`)
 - The two buttons (`renderSubmitButtonsOn:`)

The method now looks like this:

```
CcConverter>>renderFormOn: html
    html
        horizontalRule;
        form:
            [html
                div: [self renderCurrencyFieldsOn: html];
                div:
                    [html text: 'Amounts: '.
                     self renderAmountField1On: html.
                     self renderSubmitButtonsOn: html.
                     self renderAmountField2On: html]]
```

For the sake of completeness, we'll also show the definitions of the new methods:

```
CcConverter>>renderCurrencyFieldsOn: html
    html text: 'Currencies: '.
    (html TextInput)
        callback: [:input | self currency1: input];
        with: self currency1.
    (html TextInput)
        callback: [:input | self currency2: input];
        with: self currency2.
```

```
CcConverter>>renderSubmitButtonsOn: html
    (html submitButton)
```

```
callback:  
    [self amount1: (self converter  
                    amount: self amount2  
                    from: self currency2  
                    to: self currency1)];  
    with: '<---'.  
(html submitButton)  
    callback:  
        [self amount2: (self converter  
                        amount: self amount1  
                        from: self currency1  
                        to: self currency2)];  
    with: '--->'
```

```
CcConverter>>renderAmountField1On: html  
(html textField)  
    callback: [:input | self amount1:  
                (input asNumber asFixedPoint: 3)];  
    with: self amount1
```

```
CcConverter>>renderAmountField2On: html  
(html textField)  
    callback: [:input | self amount2:  
                (input asNumber asFixedPoint: 3)];  
    with: self amount2
```

16.2 Improving the Usability of the Currency Converter

When a user of the current form of the currency converter enters a currency type that doesn't exist in the program, the program aborts. If, for example, the user makes a simple typo and enters "Csech crowns" instead of "Czech crowns" and then clicks one of the submit buttons, Seaside initially has the reaction shown in Fig. 16.15. We, of course, never want our users to see that kind of an error message.

Before we turn to finding a solution to the problem, though, we should take the opportunity to determine how to find the cause when this kind of error arises. As Fig. 16.15 shows, when an error page appears in the Web browser, Seaside provides a link labelled **Debug**. When we click the link, the VisualWorks debugger opens. We

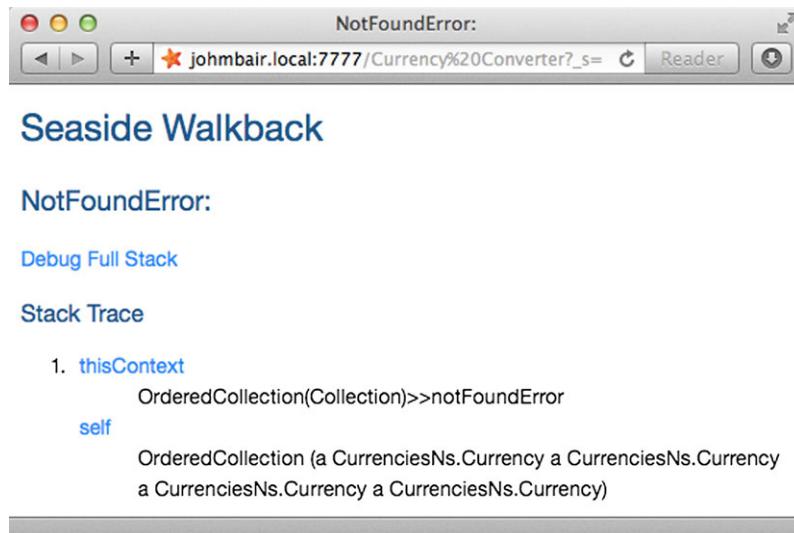


Fig. 16.15 Seaside error message for a mistaken currency input

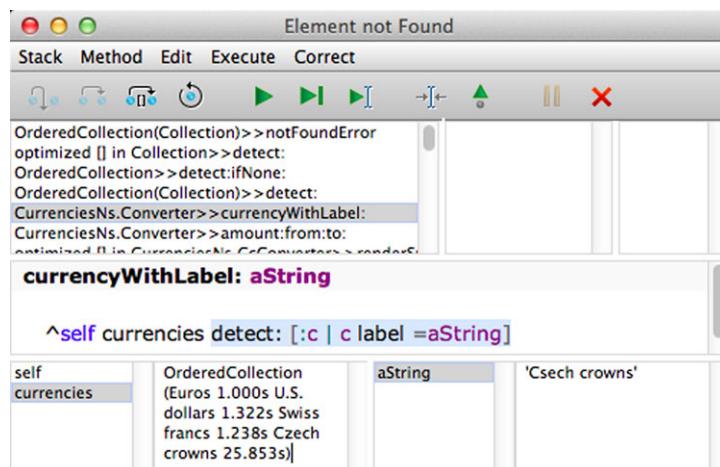


Fig. 16.16 Finding an error with the Debugger

can use the debugger in the usual way⁶ to find the error. In the method stack shown in Fig. 16.16, we find our own application listed in the first method from the top, that is, `Converter>>currencyWithLabel:`. We can see in the body of the method that the `detect:` message didn't work because the `OrderedCollection currencies` does not contain a currency with the label "Czech crowns".

⁶See Sect. 5.6.

16.2.1 Selection Lists in Seaside

Because we are permitting only currency types from a prescribed list to be input in our currency converter, it seems reasonable at this point to avoid user errors by allowing users to select from a list rather than making them key in their choice. Because such lists are available in HTML coding, Seaside can be used to create them.

Let's start by looking at the first of the two input fields for the currency labels. In the method `CcConverter>>renderCurrencyFieldsOn:`, it's been created up to now by the Smalltalk expression

```
...
(html textInput)
    callback: [:input | self currency1: input];
    with: self currency1.
...
```

In order to create a selection list at this point, we must replace the previous section with the following:

```
...
(html select)
    callback: [:input | self currency1: input];
    list: (self converter currencies
        collect: [:c| c label]);
    selected: self currency1.
...
```

The message `select` creates the selection list element. We can continue to use the callback as before. For the block parameter `input`, in this case, the element that the user selects from the list is sent. The message `list:` transfers to the selection-list element the list of character strings that are available for the user to select from. At this point, the `collect:` message computes an `OrderedCollection` with the currency labels from the `OrderedCollection` of currencies stored in the `Converter` instance. The message `selected:` tells the selection-list element which element to display as having been selected when the selection-list element is displayed for the first time.

► **Note:** As Fig. 16.17 shows, most browsers display the selection list as a dropdown menu. If you really want to show the values as a list, you can use the attribute `size`, that is, the message `size:`, to specify the size of the list, which determines the number of list elements visible at any one time.

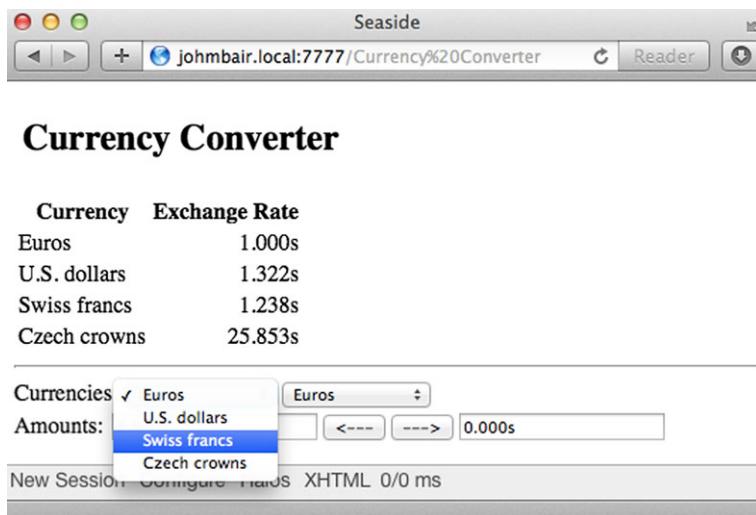


Fig. 16.17 Using a selection list to input currency types

To input the second currency label, we follow the same procedure to create a selection-list element. That results in the arrangement shown in Fig. 16.17, which appears after the user has clicked on the first dropdown-list element.

After these changes, the application performs as it did before.

16.2.2 Checking the Amount Input

At this point, we come to the second of the shortcomings listed in Sect. 16.1.6, the lack of a validity check in the two amount-input fields.

The browser always transmits an entry in an HTML input field to the server in the form of a character string. That's the reason that, in the two methods for rendering the amount fields of the input (block parameter `input`, see Sect. 16.1.7), we first sent the message `asNumber`. We can describe the way `asNumber` works in the following simplified way:

The character string is analysed from left to right. The analysis tests whether, according to Smalltalk syntax, the current character is permitted to occur in a numeric constant. The process ends either when the analysis has reached the end of the character string or when it encounters a character that may not be part of a numeric constant. A number is then formed from the characters that had been recognised as valid up to the point that the process ended.

This process does not entirely correspond to requirements for the entry test in our currency converter. For example, `asNumber` converts the character string `1,2` into the integer `1`, because Smalltalk numbers do not contain commas. For the currency converter though, that means that—should a user accidentally type a comma instead of a period as a decimal

separator—the converter will continue to calculate with whatever numbers preceded the comma without reporting an error.

► **Note:** For European applications, it would, of course, be appropriate to permit commas as decimal separators. In that event, though, we would ourselves have to program the recognition of the correct syntax for character entry. We could use a finite-state automaton, like the ones described in Sect. 11.1.4, for that purpose. For the sake of efficiency, though, we won't bother at this point.

Nevertheless, we can use a trick to use the syntax check in `asNumber`: Once `asNumber` has created a number, we use `printString` to turn the number back into a character string. If this value doesn't agree with the original, then the error must lie in the way it was input. Now we can define the following method to check the validity of a numeric input:

```
CcConverter>>amountInputCorrect: aString  
    ^aString asNumber printString = aString
```

Now we have to activate this method at an appropriate point in the program and—in case the result evaluates to `false`—issue an appropriate error message. Since the test can only occur in our Seaside application on the Web server, that is, only after the user has clicked a submit button to send an HTTP request, it must occur within a callback.

We could conceivably incorporate the test into the callbacks for the amount-input fields. It would seem to belong there naturally. Nevertheless, once an HTTP request has been submitted, the callbacks for all input fields in the form are always performed, which includes the callbacks for the two amount-input fields.

► **Note:** It's important to mention at this point that, after an HTTP request, Seaside first evaluates all callbacks for the input fields and only after that the callback for the button that activated the request.

Once all callbacks have been evaluated in this way, then the component again receives the message `renderContentOn:`.

Depending though on which of the two submit buttons was clicked, the field containing the result of the conversion will be overwritten while the other one will need to be checked; that means that it's not advisable to use this solution. For example, the output field might completely empty before the calculation, which ought not to be accepted in the input field.

Since it is only the callback of the submit button that was clicked that will ever be performed, it's practical to perform the test there. To achieve that purpose, the method `renderSubmitButtonsOn:` is expanded as follows:

```
1 CcConverter>> (renderSubmitButtonsOn: html
2   (html submitButton)
3     callback:
4       [(self amountInputCorrect: amount2)
5        ifTrue:
6          [self amount1: (self converter
7            amount: (self amount2 asNumber
8              asFixedPoint: 3)
9            from: self currency2
10           to: self currency)])
11        ifFalse: [self inform:
12          'False amount entry: '
13          , self amount2]];
14      with: '----'.
15   (html submitButton)
16   callback:
17   [(self amountInputCorrect: amount1)
18    ifTrue:
19      [self amount2: (self converter
20        amount: (self amount1 asNumber
21          asFixedPoint: 3)
22        from: self currency1
23        to: self currency2)]
24    ifFalse: [self inform:
25      'False amount entry: '
26      , self amount1]];
27   with: '--->'
```

In each of the callback blocks now, the first message sent (in lines 4 and 17) is `amountInputCorrect:`, which uses the field contents to be tested as an argument. It's important at this point to note that at this point in the program, the character string input by the user must be in instance variables `amount1` and `amount2`. For that reason, the transformation into a `FixedPoint` number must be removed from the callbacks for these input fields (see below). In this method, the transformation occurs immediately before the conversion (lines 7, 8 and 20, 21) as long as the input test found no errors.

If, on the other hand, the input test failed, the `ifFalse:` block uses the message `inform:` to issue an error message (lines 11–13 and 24–26). The `inform:` message, which is inherited from the class `WAComponent`, initiates a default dialogue; in this case, it causes the Web page for `CcConverter` to be replaced with a new one that displays the text that was sent as the argument for `inform::`. In addition, an **OK** button appears,

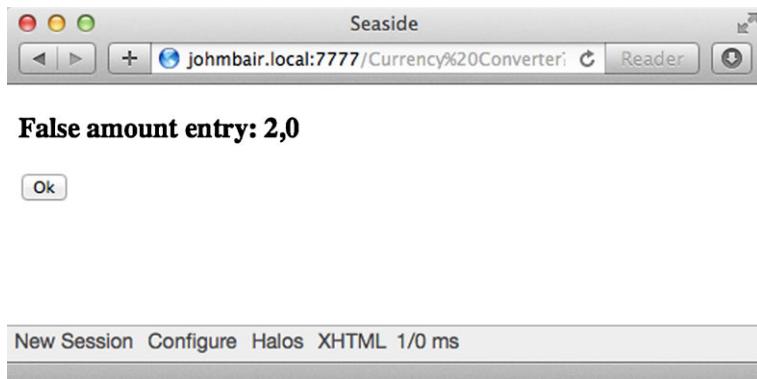


Fig. 16.18 Using `inform:` to output an error message

which the user can click to return to the application page. For example, if the user keys the character string '1,2' into the left input field and then clicks the \rightarrow button, the page shown in Fig. 16.18 appears.

After the user clicks the **OK** button, the message `renderContentOn:` will be resent to the component that invoked this dialogue, in our case our `CcConverter` component. That means that the error message disappears and the `CcConverter` page reappears in the Web browser. The incorrect entry remains in place, which the user can correct now.

As we've already mentioned, the callbacks for the two amount-input fields need to be changed in order to allow the input check to occur:

```
CcConverter>>renderAmountField1On: html
(html textView)
callback: [:input | self amount1:input];
with: self amount1
```

```
CcConverter>>renderAmountField2On: html
(html textView)
callback: [:input | self amount2:input];
with: self amount2
```

The transformation of the block parameter `input` that we had earlier used `asNumber` and `asFixedPoint` to perform is now gone.

This now completes the fix for the first two shortcomings that were mentioned in Sect. 16.1.6.

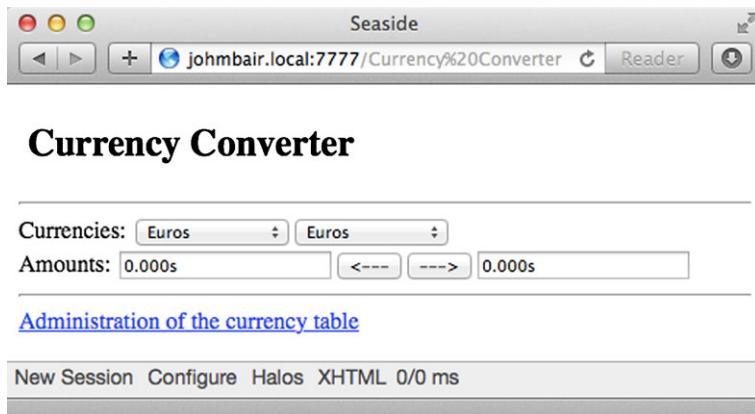


Fig. 16.19 The currency converter without the currency table

16.3 Introducing an Administrative Dialogue

The goal of this section is to expand the currency converter so that

1. New currencies can be added.
2. The exchange rates can be managed.

On our way to that goal, we'll first take the following intermediate step. We want the currency table in the upper part of the page to disappear. Now that we've introduced the dropdown lists for the currency-input fields, it's actually superfluous to display the table. We'll export the display to its own Web page, that is, to an additional Seaside component. The display will then be able to be activated by clicking a link. The main page of the application now looks like Fig. 16.19. The link **Managing the Currency Table** beneath the horizontal line now leads to the page with the currency table, which is shown in Fig. 16.20. When we click the **OK** button underneath the table, we are returned to the main page.

16.3.1 Creating the Component CcAdministration

First we need a Seaside component for the new Web page. We'll call it `CcAdministration`, and at first, we'll give it an instance variable `converter` to be able to point to instances of the model class `Converter`—just as we did for the component `CcConverter`:

```
Smalltalk.CurrenciesNs defineClass: #CcAdministration
    superclass: #{Seaside.WAComponent}
```

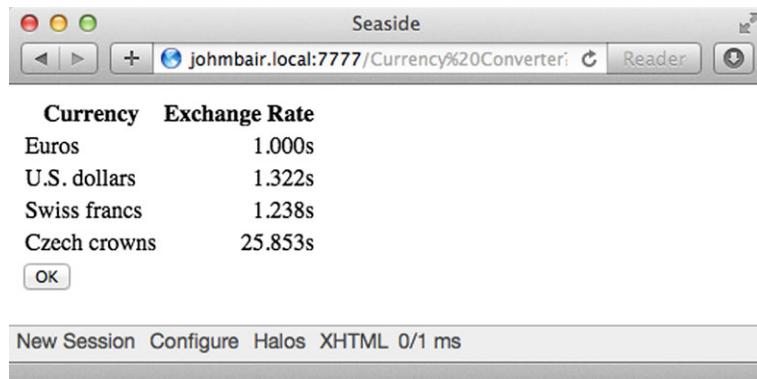


Fig. 16.20 The currency table on its own Web page

```
indexedType: #none
private: false
instanceVariableNames: 'converter'
classInstanceVariableNames: ''
imports: ''
category: ''
```

Note that, as before, the superclass is `Seaside.WAComponent`.

Unlike the component `CcConverter`, in this case, we don't need a class method `initialize` (see Sect. 16.1.4) to register this as a Seaside application on the Seaside Dispatcher page (see Fig. 16.7). The `CcAdministration` page will be available only via the link on the main page. We'll explain how to accomplish this in Sect. 16.3.2.

Nevertheless, every Seaside component needs a `renderContentOn:` method:

```
CcAdministration>>renderContentOn: html
  self renderExchangeRateTableOn: html.
  html form:
    [ (html submitButton)
      callback: [self answer];
      with: 'OK']
```

First off, this activates the method `renderExchangeRateTableOn:`, which was moved out of the class `CcConverter`, where it was no longer needed, into the class `CcAdministration`.

The callback for the **OK** button contains the expression `self answer`, which ends the component and redisplays the Web page from which the administration page had been activated. Section 16.3.3 contains a detailed explanation of how the message `answer` works.

16.3.2 Calling the Component `CcAdministration`

We want to call the component `CcAdministration` by clicking the link on the main page (see Fig. 16.19) of the component `CcConverter`, much as though it were a sub-program. The “main program” will continue running once the “subprogram” finishes. In order to be able to implement this call, the main component must first be able to “recognise” the calling component. For that purpose, we define an additional instance variable `ccAdministration` in `CcConverter` and include it in the initialise method (in the last two lines):

```
CcConverter>>initialize
super initialize.
self converter: Converter converterExample.
self
    currency1: 'Euros';
    currency2: 'Euros';
    amount1: 0.0s3;
    amount2: 0.0s3.
self ccAdministration:
    (CcAdministration new
        converter: self converter)
```

This uses the set method `converter:` to transfer the model instance (from the class `Converter`) to the newly created class `CcAdministration`, so that the administrative component can access the currency table.

We’ve already mentioned that we’ve removed the call from `renderExchangeRateTableOn:` from the method

```
CcConverter>>renderContentOn: html
html heading: 'Currency Converter'.
self renderFormOn: html.
self renderAdministrativeComponentOn: html
```

And we’ve added the call to the method

```
CcConverter>>renderAdministrativeComponentOn: html
    html horizontalRule.
    (html anchor)
        callback: [self call: self ccAdministration];
        with: 'Administration of the currency table'
```

The expression `html anchor` creates an HTML `<a>` element, which links to another Web page. Once again, the `with:` message defines the content of the element, which, in this case, is the text that appears as a link on the Web page. Something like what occurred with the Submit button, the `callback` defines what is to happen when the user clicks the link.

Clicking a link, like clicking a button, initiates an HTTP request, and Seaside performs the callback. The expression

```
self call: self ccAdministration
```

calls the Seaside component that's stored in the instance variable `ccAdministration`, which, in this case, is the instance of `CcAdministration` that was created with the `initialize` method. More precisely, “call” means that the message `renderContentOn:` is sent to the component. That causes the page to appear in the Web browser. The page belonging to the calling component becomes invisible.

This concludes the intermediate step that we defined at the beginning of Sect. 16.3, that is to say, we've moved the display of the currency table into its own component.

16.3.3 Seaside's Call/Answer Mechanism

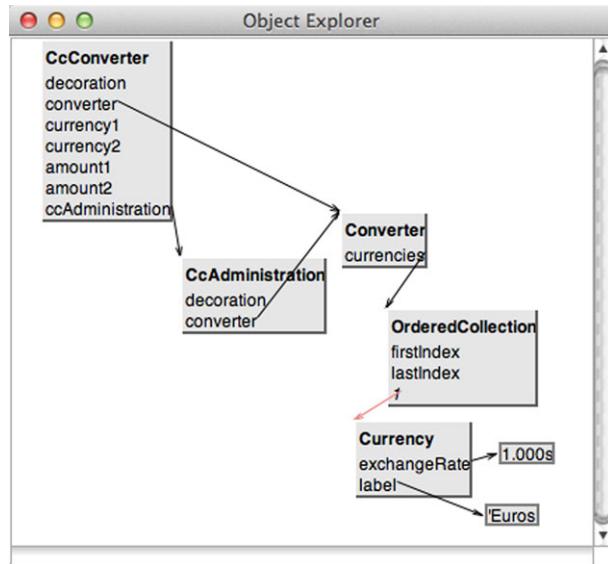
The interplay between the messages `call:` and `answer` that were used in Sect. 16.3.2 represents a central element of the Seaside framework. We'll illustrate here, using graphics this time.

If we extend the end of the `renderContentOn:` method from `CcConverter` to include the expression `self explore`, then, when we start the application, the Object Explorer⁷ from VisualWorks is also activated. After a few manipulations, Fig. 16.21 appears.

The figure shows the objects that are involved in our Seaside application and indicates how they are bound together by pointers in the instance variables. From the currency table in the `OrderedCollection` in `currencies`, only one element is shown. The image shows the static structure of the program.

⁷See on this topic Sects. 3.4.1 and A.2.

Fig. 16.21 The objects involved in the currency converter



In order to explain the dynamic process as well, Figs. 16.22 and 16.23 show schematics of the process that occur as a result of user actions. In each case, the left half of the images shows the components CcConverter and CcAdministration as they are variously embedded in the Smalltalk class library and the Seaside framework.

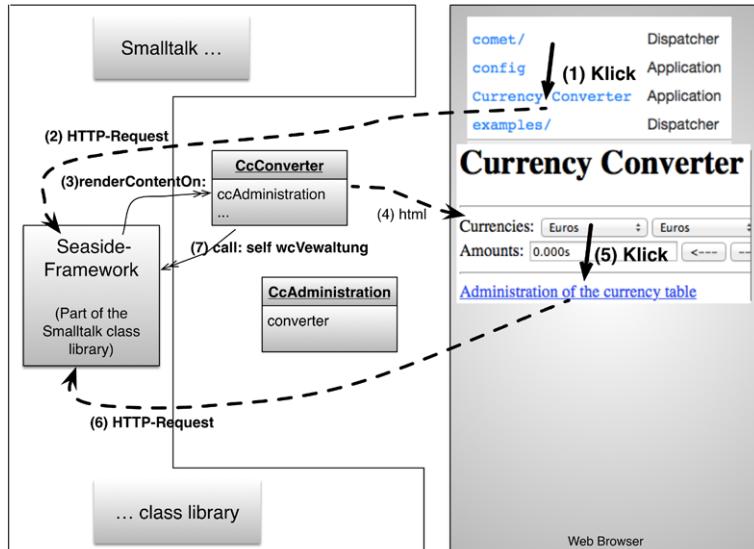


Fig. 16.22 Interplay of the components (1)

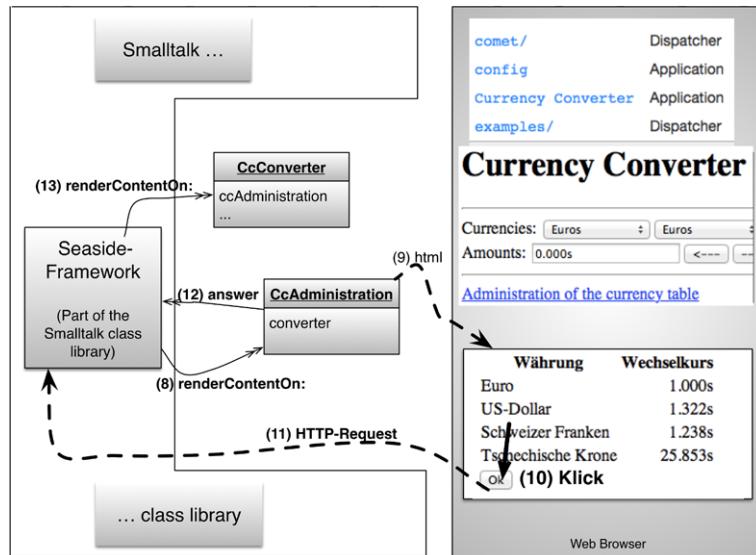


Fig. 16.23 Interplay of the components (2)

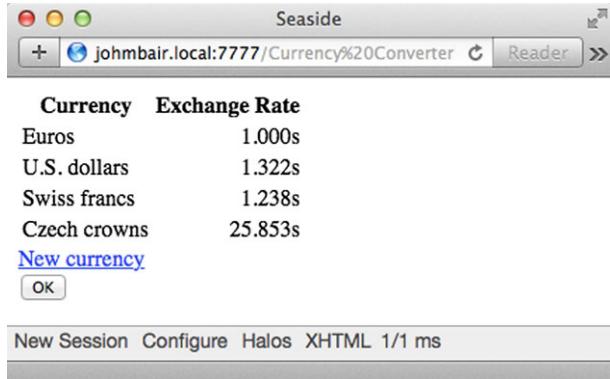
The right half of each screen shows the Web browser, although only one of the two pages is ever visible at a given moment. After Seaside starts the Dispatcher page with the link, **Currency converter** is visible. The following list describes step by step how the process proceeds:

1. The user clicks the link **Currency converter**.
2. This results in an HTTP request. Because the component **CcConverter** is connected to the link, it receives ...
3. ... from Seaside the message `renderContentOn:`.
4. Processing the `renderContentOn:` method leads (via Seaside) to the creation of the HTML elements for the currency converter page in the Web browser.
5. The user clicks the link **Managing the currency table**. This sends another ...
6. ... HTTP request.
7. Among other things, Seaside now evaluates the callback for the link, which causes the message `call: self ccAdministration` to be sent.

You can follow the remaining steps in Fig. 16.23.

8. At that point, Seaside sends the message `renderContentOn:` to the called component **CcAdministration** and notes the calling component **CcConverter**.
9. Processing `renderContentOn:` again creates HTML code, this time for the display of the currency table.
10. If the user clicks on the **OK** button on this page, another ...

Fig. 16.24 Link New currency added



11. ... HTTP request is processed.
12. The evaluations of the callbacks for the **OK** button cause the `CcAdministration` to send the message `answer`.
13. Since Seaside noted in Step 8 who the `CcAdministration` “answered,” it again sends the message `renderContentOn:` to the main component `CcConverter`.

This ends the exchange between the two components.

16.3.4 Implementing the Administration Dialogue

The next thing we want to do is to make it possible to add a new currency. We’ll use a rather simple solution. Afterwards, we’ll provide a few notes on a more elegant variation.

First, we take this method

```

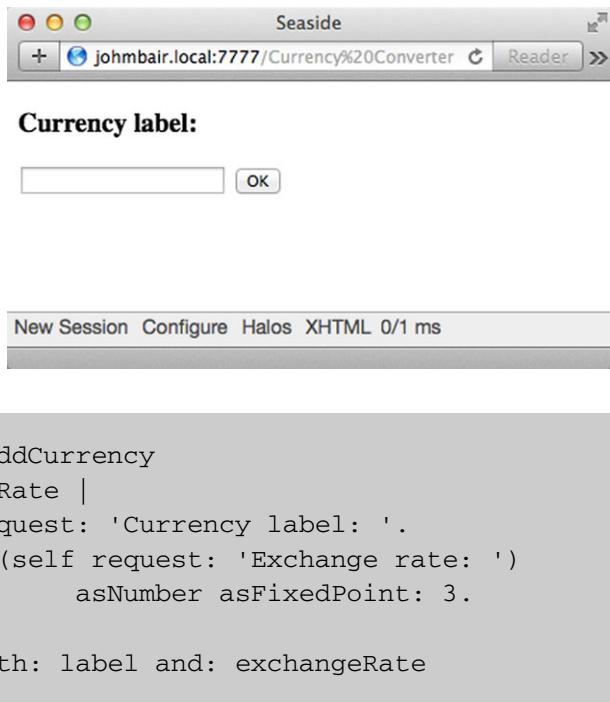
1 CcAdministration>>renderContentOn: html
2   self renderExchangeRateTableOn: html.
3   html anchor callback: [self addCurrency];
4     with: 'New currency'.
5   html break.
6   html form:
7     [ (html submitButton)
8       callback: [self answer];
9       with: 'OK']

```

and add a link anchor to it (lines 3 and 4), so that the Web page of the component now looks like Fig. 16.24.

This time, another component is not called in the callback; instead a method is activated, namely:

Fig. 16.25 Dialogue window for inputting a currency label



This method has the task of querying the user for a currency label and a rate of exchange. For this purpose, another dialogue that's embedded in Seaside is used, which is started by the message `request:`. Similarly to what happened when we used `inform:`, another window opens in which the text appears that was sent as argument. In the case of `request:`, an input field also appears. Figure 16.25 shows what happens when the partial expression

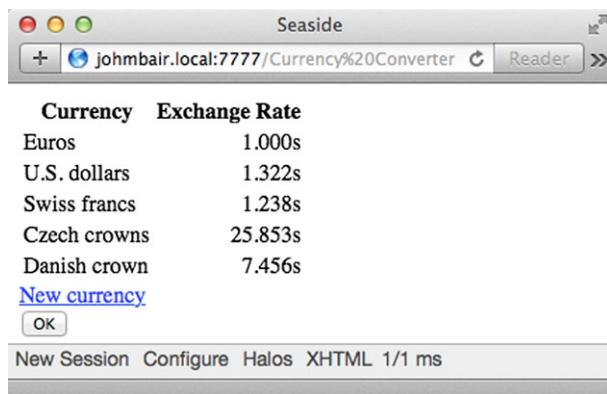
```
self request: 'Currency label: '
```

is evaluated. The character string that the user enters in the input field is the response value of the `request:` message and is stored in the local variable `label` in the method `addCurrency`.

Something very similar occurs when an exchange rate is entered after it is converted to a `FixedPoint` number. The message `addCurrencyWith:and:` sends both inputs to the model object in the instance variable `converter` (see Fig. 16.21). The matching instance method must be enhanced in the class `Converter`:

```
Converter>>addCurrencyWith: aString and: aFixedPoint
  self currencies add:
    (Currency withExchangeRate: aFixedPoint
      andLabel: aString)
```

Fig. 16.26 A new currency was added



Since the class method `withExchangeRate:andLabel:` already exists in the class `Currency`, it is used to create a new instance using the parameters that were sent, which is then added to the `OrderedCollection` in `currencies`.

Let's say that the user used the new dialogue to add the *Danish crown* with an exchange rate of 7.456 to the Euro. These values will now be added to the Web page for `CcAdministration` (see Fig. 16.26).

► **Note:** At the beginning of this section, we said that the solution we just showed was simple. This is especially true for the use of the `WACComponent>>request:` for pre-fabricated dialogues. Neither is very user-friendly, and there's still no input check for the exchange rate. A "prettier" solution might create a special component to process currencies, which could then be embedded using the Call/Answer mechanism. We leave the execution of this task to you.

Removing Currencies and Changing Exchange Rates

In order to be able to remove a currency from the table, we provide every table entry with a link by adding a column. This is done by the last four lines of the method `renderExchangeRateTableOn::`:

```
CcConverter>>renderExchangeRateTableOn: html
    html table:
        [html TableRow:
            [html
                tableHeading: 'Currency';
                tableHeading: 'Exchange Rate'].
        self converter currencies do:
            [:c |
                html TableRow:
```

Fig. 16.27 Table expanded to include a column of links

Currency	Exchange Rate
Euros	1.000s rem.
U.S. dollars	1.322s rem.
Swiss francs	1.238s rem.
Czech crowns	25.853s rem.
New currency	
<input type="button" value="OK"/>	

```
[html tableData: c label.
(html tableData)
    align: 'right';
    with: c exchangeRate.
html tableData:
[(html anchor)
callback: [self removeCurrency: c];
with: 'rem.']]]
```

Figure 16.27 shows the table expanded to include the **rem.** links. The new method

```
CcConverter>>removeCurrency: aCurrency
(self confirm:
    'Do you really want to remove the currency?')
ifTrue: [self converter removeCurrency: aCurrency]
```

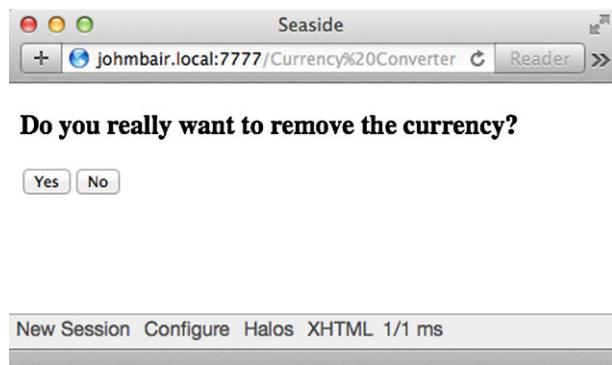
is activated in the link's callback. With `confirm:`, we once again make use of a Seaside default dialogue. Evaluating the `confirm:` message yields the Web browser window shown in Fig. 16.28. The `confirm:` message answers with `true` when the user clicks the **Yes** button, otherwise with `false`.

If the user confirms the removal, the new method

```
Converter>>removeCurrency: aCurrency
self currencies remove: aCurrency
```

is called up with the currency to be removed.

Fig. 16.28 Confirmation dialogue with `confirm:`

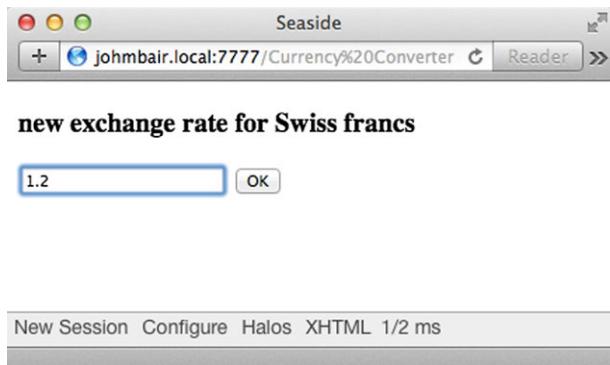


Finally, we follow a similar path for manipulating the exchange rates. First, the currency table is expanded (again in the last four lines) to include an additional link column (`edit`):

```
CcConverter>>renderExchangeRateTableOn: html
    html table:
        [html tableRow:
            [html
                tableHeading: 'Currency';
                tableHeading: 'Exchange Rate'].
        self converter currencies do:
            [:c |
            html tableRow:
                [html tableData: c label.
                (html tableData)
                    align: 'right';
                    with: c exchangeRate.
            html tableData:
                [(html anchor)
                    callback: [self removeCurrency: c];
                    with: 'rem.']..
            html tableData:
                [(html anchor)
                    callback: [self editExchangeRate: c];
                    with: 'edit']]]]
```

The message `editExchangeRate:` sent in the callback is implemented using the following method:

Fig. 16.29 Dialogue for entering a new exchange rate



```
CcAdministration>>editExchangeRate: aCurrency
| exchangeRate |
exchangeRate := (self request: 'new exchange rate for '
                  , aCurrency label)
                  asNumber asFixedPoint: 3.
self converter changeExchangeRateFor: aCurrency
                      to: exchangeRate
```

And finally we have to expand the class `Converter` to include the instance method

```
Converter>>changeExchangeRateFor: aCurrency
                           to: aFixedPoint
aCurrency exchangeRate: aFixedPoint
```

If the user now clicks, for example, the `edit` link for Swiss francs, the dialogue window shown in Fig. 16.29 appears. Then if we enter—as shown—a new exchange rate, once the dialogue is finished, the new rate will appear in the currency table as shown in Fig. 16.30.

For now, that finishes the implementation of the currency converter.

16.4 Incorporating CSS

The main purpose of this section is to show the technical possibilities that Seaside offers developers by adding Cascading Style Sheets (CSS) to their Web pages. At this point, we will not deal with design questions nor with the use of CSS in dealing with them. The Internet contains many descriptions and instructions for CSS.

Seaside offers the following basic procedures for combining CSS with HTML:

Fig. 16.30 The changed exchange rate for Swiss francs

The screenshot shows a Mac OS X-style window titled "Seaside". The URL in the address bar is "johmbair.local:7777/Currency%20Converter". The main content area displays a table of currency exchange rates:

Currency	Exchange Rate
Euros	1.000s rem. edit
U.S. dollars	1.322s rem. edit
Swiss francs	1.200s rem. edit
Czech crowns	25.853s rem. edit

Below the table are two buttons: "New currency" and "OK". At the bottom of the window, there is a menu bar with "New Session", "Configure", "Halos", "XHTML 1/1 ms".

1. A Seaside component offers an instance method called `style`. It must deliver as a response a character string with CSS commands. Like the `renderContentOn:` method, it is automatically activated by Seaside. Section 16.4.2 describes this procedure.
2. The CSS commands are stored in a separate text file. Seaside must be told how to access this file (its URL) using the instance method `updateRoot::`. The file can be
 - a. Moved to a Seaside File Library (see Sect. 16.4.3).
 - b. Located on another Web server. We will not explain here how this option works.

Our starting point for the following thoughts is the way the currency converter looks on the screen shown in Fig. 16.19. We want to use CSS commands to change the layout slightly, so that it will look like the screen in Fig. 16.31; this should make it easier to optically connect the currency and exchange rate entries with one another.

The screenshot shows a Mac OS X-style window titled "Seaside" with the URL "johmbair.local:7777/Currency%20Conv". The title bar has a magnifying glass icon and a refresh button. The main content area features a large title "Currency Converter" centered above a form. The form contains two dropdown menus labeled "Currencies:" with "Euros" selected, and two input fields labeled "Amounts:" with "0.000s" in each. Below the form is a link "Administration of the currency table". At the bottom, there is a menu bar with "New Session", "Configure", "Halos", "XHTML 0/0 ms".

Fig. 16.31 The currency converter with style sheet

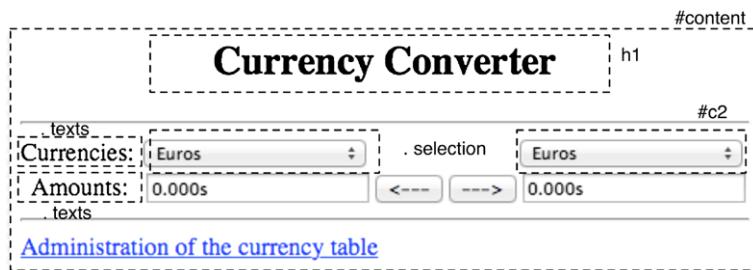


Fig. 16.32 The HTML elements and their CSS classes and IDs

In Fig. 16.32, we surrounded the HTML elements that are to be changed using CSS formatting with dotted lines. On the dotted outlines, we've marked the CSS classes and CSS IDs that we've used. The next section explains the CSS commands that have been used to make the changes:

#content This establishes the width of the entire content and places it in the center of the Browser window:

```
#content {width: 475px; margin: 0 auto;}
```

h1 The title (`<h1>` element) is to be centered:

```
h1 {text-align: center;}
```

.selection This CSS class establishes the width of the two selection-list elements:

```
.selection {width: 145px}
```

.texts This CSS class makes the fields for both screen texts the same width and right-aligns them. In addition, it sets the spacing after the right edge of the text fields:

```
.texts { float: left; text-align: right;
line-height: 1.4; width: 70px; padding-right: 10px; }
```

#c2 The selection list for the second currency is right-aligned with the input field beneath it:

```
#c2 {position: relative; left: 100px}
```

16.4.1 Combining HTML with CSS

Regardless of which of the methods mentioned in Sect. 16.4 you use to incorporate CSS text into Seaside, the affected HTML elements must be combined with the CSS classes and IDs.

The method

```
CcConverter>>renderContentOn: html
    html heading: 'Currency Converter'.
    self renderFormOn: html.
    self renderAdministrativeComponentOn: html
```

creates the entire contents of the CcConverter component. In order to use CSS to manipulate it, it must first be enclosed within a `<div>` box. After that, the message `id: 'content'` can be used to assign it to the CSS ID `#content`:

```
CcConverter>>renderContentOn: html
    (html div)
        id: 'content';
        with:
            [html heading: 'Currency Converter'.
             self renderFormOn: html.
             self renderAdministrativeComponentOn: html]
```

In HTML, that then becomes:

```
<div id="content">
    <h1>Currency Converter</h1>
    ...
</div>
```

In the method

```
1 CcConverter>>renderFormOn: html
2     html
3         horizontalRule;
4         form:
```

```
5      [html
6          div: [self renderCurrencyFieldsOn: html];
7          div:
8              [html text: 'Amounts: '.
9              self renderAmountField1On: html.
10             self renderSubmitButtonOn html.
11             self renderAmountField2On: html]]
```

the message in line 8

```
html text: 'Amounts: '.
```

must be replaced with:

```
html div class: 'texts'; with: 'Amounts:'.
```

In this case, we need another `<div>` box to use the message `class:` to assign it to a CSS class.

Finally, we need to modify the method

```
1 CcConverter>>renderCurrencyFieldsOn: html
2     (html div) class: 'texts'; with: 'Currencies: '.
3     (html select) class: 'selection';
4         callback: [:input | self currency1: input];
5         list: (self converter currencies
6                 collect: [:c | c label]);
7         selected: self currency1.
8     (html select) class: 'selection'; id: 'c2';
9         callback: [:input | self currency2: input];
10        list: (self converter currencies
11                  collect: [:c | c label]);
12        selected: self currency2
```

in order to incorporate the CSS classes `.texts` (line 2) and `.selection` (lines 3 and 8), as well as the CSS ID `#c2` (line 8).

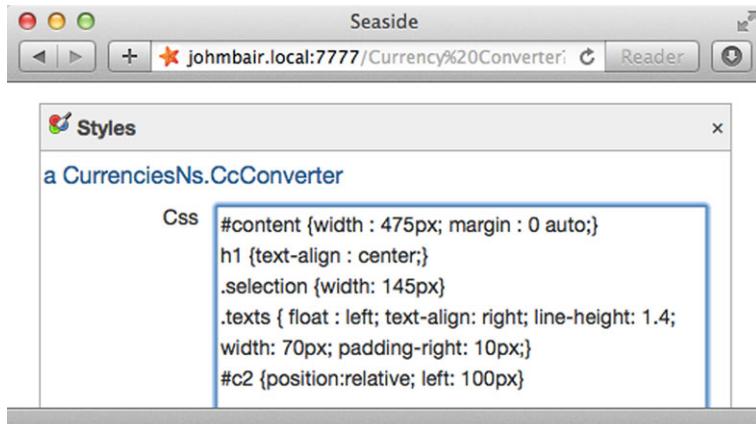


Fig. 16.33 Modifying the `style` method in the Web browser

16.4.2 Defining a `style` Method

In order to make CSS commands available for Seaside, all we have to do is provide a method called `style`, which in our case, might look like this:

```
CcConverter>>style
^
'
#content {width: 475px; margin: 0 auto;}
h1 {text-align: center;}
.selection {width: 145px}
.texts { float: left; text-align: right; line-height:
1.4;
width: 70px; padding-right: 10px;}
#c2 {position:relative; left: 100px}
'
```

The CSS commands are all within a single character string. It is returned when the method is activated (by Seaside). If the method exists and if the changes described in Sect. 16.4.1 have been made, the application appears in the Web browser looking like Fig. 16.31.

At this point, we'll point out another useful Seaside function. Once you activate the **Halos** on the Web page, you can display the CSS commands by clicking on the symbol with the associated tool-tip "CSS Style Editor". Figure 16.33 shows the result.

The advantage of using a `style` method lies in the fact that you can both display the CSS commands in the Web browser and also change them there. If you subsequently click

the **Save** button (not shown in Fig. 16.33), you can immediately observe the effects of the changes. In addition, the corresponding changes are made in the `style` method in the System browser.

An additional advantage of this procedure can also be seen in the fact that each component has its own `style` method. That means that the design can change from one component to another.

The disadvantages of the `style` method though are:

- Each time you call the method, Seaside writes the method to a file and it is made available for a call from the Web server.
- The file that is thus made available acquires a current time stamp in its name. That means that the Web browser always assumes that it's dealing with a different file. Load times are noticeably lengthened.
- The procedure is not suited for designing larger, self-contained applications.

16.4.3 Making a CSS File Available in a Seaside File Library

CSS files are part of the so-called static resources of a Web application. Those are the components of an application that are independent of the time when and location where the application is used, and also independent of the user. These might also include images (for example, logos). Under the concept of a *FileLibrary*, Seaside lets such files be integrated into the Smalltalk Image.

In HTML, access to an external file is usually managed by specifying the corresponding URLs in the `<head>` element. No `head` element, though, can be programmed in the `renderContentOn:` method. Instead, one can create an instance method called `updateRoot:`, which can be automatically activated when Seaside starts the application, and which can be used to add subelements, such as `<stylesheet>` to the `<head>` element. The `updateRoot:` method will be explained in detail later on.

First we'll show how to create a *FileLibrary* and make a file available in it. The first step is to create a subclass of `Seaside.WAFileLibrary`, which we'll call `CcFileLibrary`. Let's assume that our CSS commands are stored in a text file called `currencies.css`, and that this file is located in the same directory as our Smalltalk Image. In that case, we can load the file into the Smalltalk Image by evaluating (in a Workspace, for instance) the expression

```
CcFileLibrary addFileAt: 'currencies.css'
```

If the file is in a different directory, the path must be added before the file name.

As a consequence of evaluating this expression, the instance method `CcFileLibrary >>currenciesCss` will be created (see Fig. 16.34).

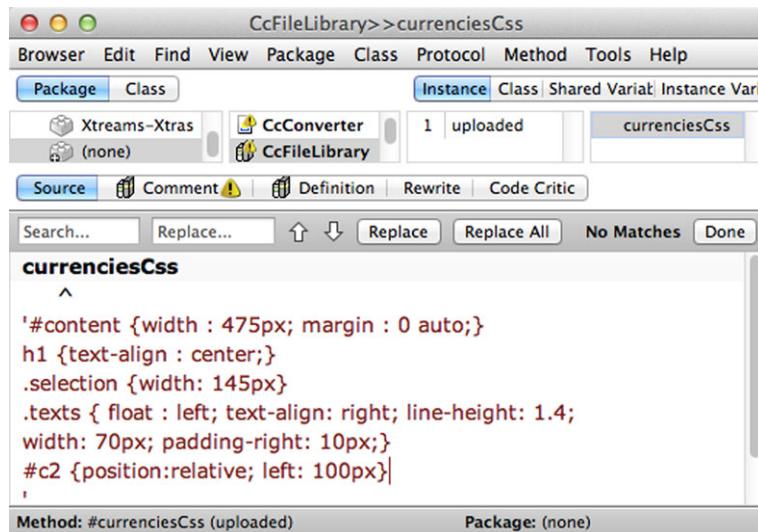


Fig. 16.34 The CSS file as a FileLibrary method

► **Note:** An anomaly of VisualWorks is that the method is initially placed—as Fig. 16.34 shows—in the pseudo package `(none)`. From there, it (or its protocol) should be uploaded to its own package using the menu command **Protocol → Move → to Package...**.

Now we still need the `updateRoot:` method that was mentioned earlier, so that Seaside can place the pointer to our CSS file and method in the `<head>` element:

```

CcConverter>>updateRoot: anHtmlRoot
super updateRoot: anHtmlRoot.
anHtmlRoot stylesheet
    url: (CcFileLibrary urlOf: #currenciesCss)

```

It's always necessary to call the method of the same name from the superclass so that the `<head>` components that Seaside creates by default remain in place. An instance of the class `SeasideWAhtmlRoot` is passed in the parameter `anHtmlRoot`, which understands, among other things, the message `stylesheet`. The object that `stylesheet` returns understands in turn the message `url:`. The message could be provided with an argument in the form of a character string containing the URL of a CSS file stored on an external Web server.⁸ In our case, the message `urlOf:`—sent to the class `CcFileLibrary`—creates a URL from the name of the CSS method.

⁸You can try it using <http://seaside.st/styles/main.css>.

Now the application should once again display the image shown in Fig. 16.31. First, though, a `style` method that might be present (see Sect. 16.4.2) should first be removed.

If we use the Web browser to display the source text for the page, we find the following `<link>` element within the `<head>` element:

```
<link href="/files/CcFileLibrary/currencies.css"
      type="text/css" rel="stylesheet">
```

The biggest advantage of using a FileLibrary is the integration of a Seaside application along with its static resources. That allows the entire Web application to be exported, as a parcel perhaps (see Appendix), and imported into a different Image. It's also easy to create a uniform design of all the components of an application, since all point to the same CSS method in their `updateRoot : method`.

16.5 Generalising the Converter

At this point, we'll discuss in greater detail some aspects of the reuse of software components. At first sight, that doesn't have much to do directly with the subject of this chapter, which is the development of Web applications. The example we've been using, though, of a currency converter, is a suitable starting point for thoughts related to reuse.

In general, the goal of reuse in software technology is the creation of modules that can be used as universally as possible, thereby avoiding the need to create copies of program code.

Our class `Converter` was developed in the first instance to convert amounts of money from one currency into another. The conversion itself occurs through the use of a constant factor. The functions programmed in the methods `Converter>>convert :` and `Converter>>convertInverse :` though, are in fact independent of whether the amounts to be converted are in fact amounts of money. The methods could just as well be used for converting units of measure for lengths, volumes, etc.

The fact that we're actually converting amounts of currency is, finally, the result of the fact that the method `Converter class>>converterExample` (see Sect. 16.1.2) has filled the `OrderedCollection currencies` with currencies and their exchange rates. If we replaced that method with

```
Converter class>>converterExample
| c |
c := self new.
(c currencies)
add: (Currency withExchangeRate: 1.0d
           andLabel: 'cubic meter');
```

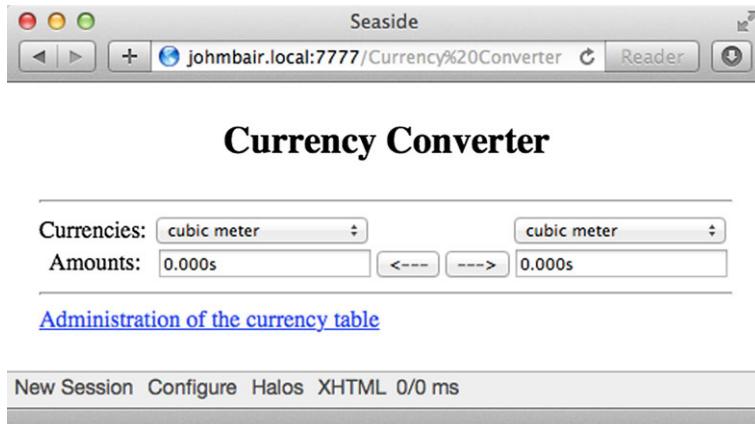


Fig. 16.35 The currency converter calculates with volumes

```

add: (Currency withExchangeRate: 1000.000d
      andLabel: 'liter');
add: (Currency withExchangeRate: 264.17205d
      andLabel: 'gallon');
add: (Currency withExchangeRate: 35.314666d
      andLabel: 'cubic foot').
^c

```

the application would convert volumes rather than amounts of money, and would require no further changes. Nevertheless, we notice immediately that the choice of label (`Currency`, `withExchangeRate`, etc.) is now out of place. While it's not important for the way the application works, it should, of course, be changed so that the label is independent of the selection of conversion amounts. At this point, though, we won't bother.

It's a more serious problem, though, that the application now looks unattractive to the user, as we see in Fig. 16.35. None of the texts on the web page are now suitable. We have to correct this.

Modification of the Classes `Converter` and `Currency`

The class method `converterExample` creates an instance of the class `Converter` with a particular dataset. In order to vary that, we could create a class method

```

Converter class>>createWithDataset: aCurrencyCollection
^self new currencies: aCurrencyCollection

```

We would pass to it as an argument a collection with instances of the class `Currency` suitable for the conversion task at hand. (As we've already mentioned, it would be advisable for the readability of the program to rename the class.) The dataset would then be created outside of the class `Converter`. That would allow the class to be very flexibly deployed.

For the sake of simplicity, in the following explanations, the method `Converter class>>converterExample` from Sect. 16.1.2 will be renamed `converterExampleCurrency` and a second dataset will be created, for which a method `converterExampleVolume` has been created.

Modifying the Class `CcConverter`

For one thing, we're going to expand the functionality of the component so that the user will be able to select from among different conversion categories. For another, we'll rename the text on the Web page so that it will either have general validity or else will be able to be varied.

For that purpose, we'll first define two new instance variables in the class `CcConverter`:

categories contains a dictionary with the character strings '`Currencies`', '`Volumes`', etc. as keywords and suitable instances from `Converter` as values.
category contains the keyword for the current conversion category, which is selected by the user.

In addition, we will define the `initialize` method for the components as follows:

```
CcConverter>>initialize
super initialize.
self categories: Dictionary new.
self categories
    at: 'Currencies'
        put: (Converter converterExampleCurrency).
self categories
    at: 'Volumes'
        put: (Converter converterExampleVolume).
self category: 'Currencies'.
self converter: (self categories at: self category).
self ccAdministration: (CcAdministration new
                        converter: self converter)
```

One `Converter` instance for currency conversion and another for converting volumes are entered in the dictionary `categories`. The variable `category` receives the character string '`Currencies`' and the variable `converter` receives the appropriate



Fig. 16.36 Generalised appearance of the converter

Converter instance as initial values. That configures the application for its initial start for currency conversion.

In the `renderContentOn:` method for `CcConverter`, at first only the title (`<h1>` element) will be generalised.

In the method `renderCurrencyFieldsOn:` from Sect. 16.4.1, the `<div>` box that creates the fixed text “Currencies” is changed as follows:

```
(html div)
  class: 'texts';
  with: self category, ':'.
```

That causes the text to depend on the content of the instance variable `category`.

The link text “Managing the currency table” that we defined in the method `renderAdministrativeComponentsOn:` (see Sect. 16.3.2) can now be changed into, say, “Managing the dataset”.

Figure 16.36 shows the component `CcConverter` in the Web browser after having made these changes.

Now we just need an option for the user to be able to change the conversion categories. This will be accomplished through an additional selection list. First of all, the method

```
1 CcConverter>>renderContentOn: html
2   (html div)
3     id: 'content';
4     with:
```

```

5      [html heading: 'Converter'.
6      self renderSelectCategoriesOn: html.
7      self renderFormOn: html.
8      self renderAdministrativeComponentOn: html]

```

is expanded in line 6 by calling the method `renderSelectCategoriesOn:`. In this method, then the selection list is created:

```

1 CcConverter>>renderSelectCategoriesOn: html
2   html form:
3     [html div: 'What do you want to convert?'.
4      (html select)
5        list: self categories keys;
6        selected: self category;
7        callback:
8          [:input | self category: input].
9        (html submitButton)
10       callback:
11         [self converter:
12           (self categories at: category)];
13       with: 'ok']

```

We've used the keywords in the dictionary `categories` (line 5) for the values in the selection list. The category that the user selects is written in the callback in the variable `category` (line 7). In the callback for the submit buttons (lines 9 and 10), the variable `converter` is then bound to the `Converter` instance that corresponds to the selected category.

Figure 16.37 shows an application example of the converter after selection of the category “volumes”.

► **Note:** The URL shown in Figs. 16.36 and 16.37 continues to contain “CurrencyConverter” as part of the address. That is, because we haven't yet adapted the class method `initialize` from the class `CcConverter` (see Sect. 16.1.4). We could, however, change the argument in the message `registerAsApplication:` to something like '`Converter`'.

We've now generalised our application to an extent that we could add additional conversion categories to it with little effort. All we need to do is add an additional dataset in



Fig. 16.37 Converting units of volume

the form of a “converterExample...” method and appropriately expand the initialisation of the dictionary categories.

Instead though, we could store the categories dataset in an external text file, which would then have to be read in during the initialisation phase of the application; in that case, changing the dataset would require no additional programming changes.

The example we’ve shown here only light sketches the topic of reusing software components. The construction of reusable modules of as high a quality as possible is a complex software engineering task that has no single recipe for success. It also happens sometimes that parcelling out components so that they can be reused leads to a significant increase in their complexity. In such cases, it’s necessary to perform a cost-benefit analysis, which may end up finally being decided by asking to what extent a component is expected to be reused. Particularly for the design of graphic interfaces, though, it’s quite typical that there are frequently recurring representational tasks. For that reason, Seaside contains several subclasses of `WAComponent` for special application cases. As an example, we’ll mention the class `WATableReport`, which provides a simple way to display a collection of objects in a Web browser in the form of a table.

16.6 Further Seaside Concepts

The previous sections have been able to provide only a glimpse into the basic concepts that Seaside provides for developing Web applications in Smalltalk. We can’t describe here in detail other useful techniques that are also necessary for the design of modern GUIs. Instead, we’ll use this section to provide a brief summary.

The sample application whose development was described in this chapter made use of only a few HTML elements. In fact, though, Seaside provides messages for creating nearly all HTML elements. You can find a good summary in Sect. 7.6, “Learning Canvas and Brush APIs” in Ducasse et al. (2010).

Sect. 16.3 showed how one component can call another by sending a `call:` message. As a result of such a call, the page of the calling component is replaced in the Web browser by that of the called component. Sometimes, though, we may want to see the contents of both components on the page. For that purpose, Seaside provides the concept of embedding components. (See Chap. 12 in Ducasse et al. (2010).)

For some Web applications, it’s necessary to manage user-specific data. For example, it’s sometimes necessary for users to identify themselves (or log themselves in) before they can use an application. Seaside provides the concept of *Sessions* for that kind of information. The relevant class is called `Seaside.WASession`. (See Chap. 18 in Ducasse et al. (2010).) Sessions are also used to manage user-specific resources such as the access to databases. In these cases, it’s also especially important to terminate sessions properly and thus release resources that are no longer needed.

Our converter still leaves a lot to be desired in terms of user-friendliness. A user of a modern GUI would certainly expect today that input in one of the amount fields would immediately result in an update of the contents of the other amount field. In that case, we could do away with the two submit buttons used to bring about the calculation. In HTML, however, HTTP requests can only be initiated by using buttons or links, not by entering input in text fields. Without the submit buttons, the calculation could only be initiated through additional programming logic, which would run in the Web browser. Programs written in JavaScript would be a solution. Seaside uses *Prototype*,⁹ together with `script.aculo.us`¹⁰ and `jQuery`¹¹ to support the incorporation of two JavaScript libraries.

One goal in the development of a Web application can be making it available on a commercially available Web server such as Apache. You can find some information on deploying a Seaside application in Chap. 23 of Ducasse et al. (2010).

Another task that we haven’t looked at here is saving data from an application to a database. Chapter 25 in Ducasse et al. (2010) provides a summary of various technologies that can be used from within Seaside for the long-term storage of application data.

The concepts we’ve enumerated here are in no way a complete list of the possibilities that Seaside offers for the design of Web applications. You can find additional information at Seaside (2013). Another good source of assistance can be found in the samples delivered along with Seaside (see Appendix A.3).

⁹<http://prototypejs.org>.

¹⁰<http://script.aculo.us>.

¹¹<http://jquery.com>.

In this book, we've been able to tell you about only the basic concepts of object-oriented programming and elementary techniques for creating Smalltalk programs. And so there are several topics that you'll have to grapple with to be able to do a good job of developing object-oriented applications. At this point, we'll provide you with a few tips for finding information on these topics. But you must always keep in mind that you can't learn programming from books. Books can only provide a little help, which can never replace practical experience.

Designing Class Schemas

One of the central questions in the development of object-oriented applications is this: How can you find a class schema that suitably reflects the slice of the real world that the application represents?

We can find answers to this question in the phases of the application-development process that occur before the actual programming. These phases are referred to as *object-oriented analysis* and *object-oriented design*. These topics are described in the kind of books that frequently have titles that contain phrases like *software engineering*, *software technology* or *object-oriented software development*. There's a great deal of literature on this subject. As a sample for many others, we'll recommend that you read Balzert (2001), Booch et al. (2007) and Oestereich (2005). These books describe systematic methods of proceeding to analyse the potential subject of an application so that it can subsequently be adequately described through a system of classes. The case studies in this book were able to handle this theme only in a rudimentary way.

Actually, the object-orientation paradigm reveals its greatest effect precisely in the analysis and planning phases. Looking at a slice of the real world as a system of interacting objects, and then reconstructing that slice in a technical model—that is the basic thought behind object-orientation. In practice, this way of thinking has proven successful. Compared to those initial phases, object-oriented implementation can be regarded as being of almost secondary importance. That also applies to the choice of programming languages.

Because of its simplicity, transparent structure and particularly “pure” implementation of the basic object-oriented idea, Smalltalk offers itself as a logical choice.

Design Patterns

For quite a while, so-called *design patterns* have been gaining importance in the field of software engineering. The concept refers to standard solutions for problems that frequently occur in programming. Using design patterns is an attempt to counter the tendency to “reinvent the wheel” that one often encounters in software development. An example of such a problem is using a tree structure to model complex, real objects. Think of an automobile as the kind of complex object we’re talking about. A tree structure is intended to describe the Isa relationships of the components that make up an automobile. The complete car is at the root of the tree. It consists—stated a little simply—of the component’s body, chassis and drive train; the drive train, in turn, consists of the engine and transmission. On the leaves of the tree, we find the parts that no longer have component parts, for example, the tailpipe. Experience teaches us that similar structures occur in many contexts. Among the design patterns there is one called *Composite*, which describes an object-oriented standard solution to this problem. We get the name of the Composite pattern from the standard source on this topic (Gamma et al. 1995).

That book describes at least 20 different design patterns from the perspective of the C++ programming language. Alpert et al. (1998), relying heavily on Gamma et al. (1995), shows how design patterns are implemented in Smalltalk.

These design patterns represent a medium level of abstraction. On a somewhat higher level, we find so-called architectural models that include, for example, the model-view-control paradigm that we described in Sect. 14.4. It also describes a kind of model solution for the construction of an interactive, object-oriented program, which, in turn, several of the design patterns described in Gamma et al. (1995) are used.

In contrast, so-called *micropatterns* are concerned with the elementary question of the type asking what to call classes, methods and variables and how to format while software ergonomics looks at questions on fitting the add comments to program code or how to construct a `printOn:` method. *Composed method* (see Sect. 14.3) is also a micropattern. An important collection for Smalltalk programmers can be found in Beck (1997).

Graphical User Interfaces

Among the aspects of programming that are specific to Smalltalk that we did not cover in this book, we find the topic mentioned briefly in Sect. 14.4 relating to the programming of native graphical user interfaces (GUIs) for interactive applications. For more extensive technical information, we recommend consulting the documentation of the Smalltalk development system being used. For VisualWorks, the documents Cincom Systems (2011) and Cincom Systems (2014a) are particularly important, which are delivered as PDF files as part of the product’s online documentation. Hopkins and Horan (1995) also provide a good introduction to implementing the model-view-control paradigm in VisualWorks.

When designing GUIs, though, ergonomic and psychological considerations are at least as important as technical questions. This is the topic that the field of study called *software ergonomics* concerns itself with (see, for example, Jacko 2012). Software ergonomics looks at questions of suiting the interaction between humans and computers to human processes and capabilities. Starting with discoveries in cognitive psychology, Preim (1999) draws conclusions about designing interactive systems.

Free-Standing Smalltalk Applications

We've gotten to know Smalltalk programming as a process in which the development environment is used to supplement an existing class library with application-specific classes. In order to activate the methods of the application, it's always necessary to evaluate Smalltalk expressions (in the Workspace, for example). That means that one's own application runs using the tools of the development environment. Nor does anything change fundamentally when a GUI is added to the application.

When a finished application is ready to be delivered to a customer though, we don't usually want to deliver the development environment too. In fact, it's possible that one may not deliver it for licensing reasons.

Nevertheless, the classes for one's own application as well as the classes that they need from the original class library (along with the original class library's other classes, including those that are required only by the development environment) are all located on one and the same Image. It's not an insignificant task though to remove all of the classes from this "jungle" that are not needed for the application. It's almost impossible to do without the aid of a tool. The way such a tool functions and how it is to be used depends strongly on the development system that was used, even though the basic principle—that is, how to remove unneeded classes from the Image—is always the same. VisualWorks offers a tool called *Runtime Packager* that supports the programmer in performing this task. Cincom Systems (2014b) describes how to use the tool.

Developing Smalltalk Applications in Teams

Professional development systems must make a mechanism available that allows a team of developers to cooperate in an orderly fashion on the development of an application. For one thing, it's necessary to permit various programmers to work as independently as possible on the development of application components. For another, though, these independently developed components must be able to be united into a total system. One problem with this way of working is that various versions of the components are created during the development process, which work together only within certain configurations. For testing purposes, though, the components are constantly in need of being brought together. And completed applications that have been delivered to customers also exist in different versions. So-called *configuration management* has the task of managing compatible versions of either the components or of the entire application.

In a Smalltalk environment, one can assume that each individual programmer works on a "private" Image, to which are added the program components (classes, methods) that the

programmer develops. Nevertheless, programmers must always take other programmers' components into consideration. The classes developed by other programmers must therefore be available to other programmers, and they must be able to add them to their own Images if it's necessary, say, to perform tests.

The simplest method that's available in VisualWorks for doing this is to export a method, protocol, class or class category to a text file. For example, the context menu for each field contains the item **File Out As...**. The *File Browser* tool can then be used to import those text files into an Image. But this way of working is certainly sufficient only for very small teams, because exchanging components requires close cooperation among team members; moreover, there's no kind of version control.

VisualWorks offers a tool that programmers can use to manage within an Image—or just keep straight—the components of various applications that they might be working on in parallel. This tool is the so-called *Change-sets*, the use of which is described in Cincom Systems (2014e).

VisualWorks offers a complete configuration-management system that includes the add-on component *Store* (see Cincom Systems 2014d) and provides both version control and an idea for application components called *Packages*, which we've already encountered in Sect. 5.7. These packages are the units in which Store manages the program code. Store uses a commercial relational database to store these components independent of an Image.

By the way, VisualWorks also offers add-on components that make it possible to develop Smalltalk applications that can access relational databases. (See Cincom Systems (2014c).)

Additional Sources of Information

The Internet offers countless opportunities for finding information about Smalltalk in general and about special development systems in particular.

First of all, we'll mention Cincom's Smalltalk website. At

<http://www.cincomsmalltalk.com>

you can find information about all of Cincom's Smalltalk products. This site also allows you to download the non-commercial version of VisualWorks for free.

You can find a good first step into the topic of Smalltalk at

www.smalltalk.org
www.stic.st

Both sites contain, among other things, a great number of links to additional sources of information about

- Smalltalk projects
- Commercial and free Smalltalk development environments
- Commercial Smalltalk applications
- User groups

- Comparisons with other object-oriented programming languages
- Smalltalk courses

For those of you who want to learn Smalltalk, the website of my colleague Professor Dr. Stéphane Ducasse bears mentioning:

`stephane.ducasse.free.fr`

Along with much other interesting information, the site also makes available online a collection of older, out-of-print Smalltalk books.

You can find material to accompany this book at the author's Web page:

`brauer.nordakademie.de`

In previous chapters, we learned how to develop software applications. From the outset, we have used the currently most utilised development paradigm, “object-orientation” (OO). Alan Kay and a group of scientists at Xerox’s Palo Alto Research Centre came up with this methodology in the 1970s.

During the years of mainframes and midrange systems, the goal had been to create programmable electronic devices for everyone’s personal use. One essential prerequisite for achieving this was to make programming easy for non-technical people. The team of scientists realised that programming doesn’t mean to teach a machine to do something; rather it is capturing a model of the reality. How it’s made to run on the hardware is just a technical detail that is of no concern to the user. Team members worked with children to understand how they learn about the world, which gave them ideas for the new concepts. They implemented an environment that captured the ideas of object-orientation and the philosophy behind it in a pure and simple way and called it “Smalltalk”.

Learning is made most efficient by understanding the concepts and applying them. Throughout the book, we used the Smalltalk environment to experiment with and understand the basics of “OO”. Its simplicity, openness and dynamic nature with immediate feedback gave us the power and means to gain that insight.

Yet, Smalltalk isn’t an “academic toy”; it’s a proven development and runtime environment for production systems of all kinds. The exact same characteristics that make Smalltalk a good learning environment also make it perfect for professional use. Its simplicity allows easy modeling of even the very complex aspects of the domain being built right into the code. The dynamic system with immediate feedback enables exploratory understanding of the problem at hand and incremental implementation of its solution. Its openness enables better understanding and adaptation of the system to one’s own needs. Together, this leads to a highly efficient workflow that allows users to build better applications in less time and with less people. And most importantly, the developed application can easily be modified to accommodate changed or new requirements.

Today, organisations of all kinds and sizes from all over the world use Smalltalk to implement business-critical applications. Each of the examples that we provide below have one thing in common: They each examine how companies have brought value to their users by providing better functionality, quicker time-to-market and the flexibility to meet any—even unforeseen—needs.

18.1 AG5—Safety Compliance

AG5 uses Smalltalk to develop software for professional groups that require fast, capable, error-free actions: ministries, safety and rescue organisations such as ambulance personnel and firefighters as well as large international firms that have their own fire departments such as the chemical, pharmaceutical, energy and food industries. The Dutch company enables those safety entities to understand and support the functions, experiences and expertise of their staff.

AG5's skill management software allows the management of daily scheduling of all workforces. Further aspects covered are the financial compensation and administrative processes of education, training and deployment for multi-disciplinary crews and individual specialists alike.

As it is absolutely mandatory to ensure suitable responses to major incidents, clients appreciate the software's unique ability to share organisational data and support organisational processes both within one entity and between different organisations.

The software is capable of being tailored to each sector's legal requirements and an organisation's individual demands, making it highly functional across a variety of industries. In order to outline the customer's needs and expectations, AG5 usually begins the deployment process by interviewing the customer. Capturing and storing of personal data, complex organisational structures and hierarchies provide a clear understanding of the skills of each employee, and this task is easily accomplished thanks to the solution's flexibility.

It's obvious that due to the nature of the application areas, the system must be extremely reliable, very efficient and capable of being up and running 24×7 . The system, which is 100 percent cloud and completely web-based, provides access to up-to-date information at any time and from any computer around the world. Sensitive information is encrypted by AG5 and stored and monitored on their local servers.

AG5 was created in 2005, and after experiencing steady revenue growth, it has been profitable since 2007. With more than 10,000 paying user accounts on a recurring subscription-based license, several hundreds of concurrent users are served daily by AG5-owned hardware.

In today's ever-changing environment, Cincom VisualWorks and its design principles provide the ability for AG5's development team to be extremely productive. Since they are capable of instantly adopting new training methods, increasing legislation requirements and changing organisational structures, they are rapidly achieving a distinct advantage over their competition, which helps them to quickly and efficiently respond to new, challenging demands.

18.2 Cognitone—*Music*

The German software company, Cognitone, is the maker of Synfire, a sophisticated software package for composers, songwriters and music producers.

Just as a canvas, brush and paint don't create a painting without the painter's ideas, the latest and greatest sound studio equipment doesn't make music. The composer must conceive composition ideas such as rhythms, harmonies, melodies and instrumentation, before production begins. Cognitone supports the musician from his or her initial idea to the realisation of a composition. Based on the techniques of artificial intelligence (AI), the technology is able to understand and manipulate music at a high level of abstraction, setting it apart from most existing standard recording and mixing software. Synfire makes ideas visible and comprehensible before they can be heard. Cognitone coined the term "music prototyping", which describes this new explorative and dynamic workflow.

With its strengths in music prototyping, the product actively supports the development of original compositions from the ground up. Users increase their productivity and appreciate the inspiration that Synfire evokes for new musical ideas. It is used for movie scoring, game soundtracks, television commercials, contemporary orchestral works, electronic music, rock, R&B, pop, Latin, jazz and songwriting of all styles.

Several years of research and experimentation were spent to understand the requirements, model the domain, explore AI algorithms and design an intuitive yet powerful interface for workflows that hadn't existed in other tools that were available for the market. Most of this was accomplished immediately in the Smalltalk environment. Large parts of the product had to be refactored or rewritten from scratch multiple times. Despite this huge challenge, the founder and CEO of Cognitone, André Schnoor, considers this work to be a lot of fun. In fact, it prompted him to invent the phrase "development at the speed of thought" that hadn't been possible without the lean and expressive Smalltalk language and its powerful development environment.

Cognitone assumes that due to its limited resources, the product would have failed, if they had started with C++, Objective C or Java, the standards in their business. Among the available Smalltalk offerings, they selected Cincom VisualWorks for its performance, maturity and cross-platform coverage. Cognitone's software is currently available on OS X and Windows.

18.3 Georg Heeg eK—*Software Projects and Services*

For over 25 years, the German company Georg Heeg eK has been developing software solutions and providing Smalltalk-related services to national and international companies in many different industries and tackling a vast variety of business problems. Not afraid of a challenge, the company's founder and president enthusiastically sponsors cultural initiatives, which has led him to take on two "unusual" projects in his hometown of Köthen.

Although it was a well-known fact that the German composer and musician Johann Sebastian Bach lived and worked in Köthen from 1717 to 1723, no one knew exactly where Bach had resided. The local government and the Bach Society of Köthen had the goal to increase tourism. Therefore, they initiated a project that would once and for all answer this question. The study, funded by the State of Saxony-Anhalt and the European Commission, required extensive research as well as sophisticated technology. For the technology side of the project, Georg Heeg eK was hired.

The first step was to have the Bach experts assemble all of the available historical data, ancient tax records, cadastre lists, letters, etc. The Heeg team built a database that held this data in an electronic format for further processing. In order to narrow down the set of possible houses where Bach might have resided, they added an application that allowed them to capture dependencies from which to build and visualise a semantic network of the data in various iterative steps. They wanted an integral, seamless system, so they chose the object-oriented Smalltalk database GemStone/S for the data layer, Cincom VisualWorks for the logic layer and the Smalltalk web application framework Seaside as the presentation mechanism, which enabled the researchers to have web access to the data.

At the end of the research, two houses remained, and the historians agreed that Bach must have lived in both. The first house where he lived from 1717 to 1719 had been torn down 50 years ago. However, the other building where he resided for the rest of his stay in Köthen is still standing, and it is now a designated landmark and a stop for tourists.

The team of software developers and historians credits the success of the time-constrained project to the rapid feedback that was brought about by Smalltalk, which aided in their constant search for new information and the immediate evaluation of new possibilities. If any of these capabilities had not been available, the progress of the exploration could have been compromised.

A second, independent project was the construction of an exhibit at the Köthen Castle. The interactive exhibit, called “Erlebniswelt Deutsche Sprache”, showcases the extensive history of the German language. Because it was scheduled to open in just 3 1/2 weeks, some people as well as local news reports said that trying to accomplish such a feat in that amount of time was “crazy”. However, the exhibition was indeed finished on time and with very positive reviews.

The multimedia environment lets visitors experience and learn about all aspects of the German language in a playful way. The interactive displays as well as the supporting infrastructure are realised using Cincom VisualWorks. All content is presented with a common look and feel by means of embedded Web applications that were developed using seaBreeze—a Smalltalk WYSIWYG IDE for Web applications. One of these applications automatically gathers information found on various Internet sites and compiles a new Web page for display. Another application controls the cinema by determining the videos to be shown with their viewing times, automatically dimming the lights before the videos start and turning them back on upon completion. The same technology was used to celebrate the 400th birthday of August of Saxony at the Moritzburg art museum in Halle (Saale).

Georg Heeg’s team enjoyed taking on these “unusual” challenges and is proud that the projects were completed on time and with great success.

18.4 JPMC—*Financial Services*

JPMorgan Chase is a leader in the financial services industry with assets totalling \$2.4 trillion. A component of the Dow Jones Industrial Average, it serves global capital markets by providing expertise in the areas of investment banking and management, research, private banking, equities, treasury and security services.

The company has developed its own financial risk management and pricing system known as Kapital. The system, which was developed using Cincom VisualWorks, enables hybrid derivative businesses to trade highly complex financial instruments in large quantities. JPMorgan Chase and the Kapital system overshadow the competition by being able to provide faster time-to-market and exceptional scalability. According to David Ramel in the magazine Application Development Trends, in the highly competitive investment banking market, time-to-market is critical, and “milliseconds make millions”.

The seamless integration between the financial application framework and the technical infrastructure that Smalltalk delivers enables IT and business staff to develop and prototype new ideas and concepts and incorporate them directly into the product. Now, those who are creating new products are no longer required to completely comprehend how the fundamental infrastructure works, which lets financial experts spend more time on their area of expertise: finance. There’s no need for them to know the intricate complexities of such areas as database and memory management, distribution algorithms, etc. The result is extremely quick turnaround for a new financial product.

Many financial institutions are creating new complex deals by using spreadsheets or other semi-automated approaches to constructing derivative trading models. Yes, it is possible to develop highly complex trades in this way. However, due to the large amounts of complex, computing intensive financial information that needs to be processed, it’s a cumbersome process that severely hinders the ability to complete substantial quantities of these trades. Thanks to Kapital’s rich financial and infrastructural development tools, developers can identify potential bottlenecks and remedy the situation quickly and easily. With Kapital, scalability issues don’t exist—even with high volumes—which makes it possible for JPMorgan Chase to surpass the competition.

The Kapital system invokes many capabilities and benefits. In the area of IT, Kapital:

- Supplies a framework that enables the construction of new financial applications.
- Provides a complete development environment with debugging capabilities and source code management.
- Enables you to store financial objects of any type due to it being a transparent persistence mechanism.
- Allows you to distribute any type of financial task by supplying a distribution infrastructure.

It’s been said that “fashions are seasonal”, and this statement also pertains to “fashion programming”. JPMorgan Chase knew that Java, C++ and C# would not satisfy all of their requirements and needs. The productivity that Smalltalk has delivered has more than

justified its use, and the company appreciates the fact that development resources don't need to be Smalltalk "experts". Another fact that explains Smalltalk's use is the realisation that had they built Kapital using another language, it would have cost them three times as much in terms of resources for application development and maintenance. Kapital also meets their portability and stability requirements, which are crucial. The base system is capable of running on three operating systems including Solaris, Windows and Linux. The process for changing the primary infrastructure can be time-consuming. Most of the time is dedicated to recompiling and retesting external "C" calls whereas the Smalltalk portion requires very little modification effort. By using Smalltalk, much less time is required for JP Morgan Chase to adapt to new technologies, and they realise that this would not have been possible using a different language.

Currently, the Kapital development team makes up less than 1 percent of JPMorgan Chase's entire staff. However, a very large percentage of the company's total revenues are due to the businesses that Kapital supports.

18.5 Key Technology—*Manufacturing*

With its headquarters in the USA and facilities in the Netherlands, Belgium, Australia and Mexico, Key Technology offers automated process technology that assures quality, efficiency and rapid return on investment for food, tobacco, pet food, plastics and pharmaceutical/"nutraceutical" (derived from nutrition and pharmaceutical) manufacturing companies.

The simple principle of quality assurance, not only in the food industry, is "the good ones go into the pot, the bad ones go ...". Yet, as usual, the devil is in the details: what is "good" and what is "bad"? How can we separate them? In masses? Here is where Key's solutions come in. They have changed the way food is inspected, sorted, handled and prepared for processing. The ultimate goal of Key's solutions is to maximise the yield of "good ones" because for Key's customers, maximum yield means maximum cash.

In the mid-1990s, when Key was looking for a suitable technology for its next generation food sorters, they had one major requirement: it had to be future-safe, enabling fast continuous development as well as disruptive innovation. From a business perspective, this would ensure their ability to keep and expand their leading market position. Of course, technical demands such as good performance, high scalability, proven robustness, easy maintainability and cross-platform support also had to be met. After considering all aspects and evaluating the different options, Key determined that Smalltalk was the "best match". Upon comparing Smalltalk dialects, Key found that its requirements were best met by Cincom VisualWorks, which they use to implement the operating systems and user interfaces of their sorters.

The first product shipped was Tegra, which revolutionised the world of automated optical sorting. It is still recognised worldwide as the industry's most accurate in-air defect-removal system. With a 360-degree view and by combining visual, infrared and ultraviolet

data, Tegra is able to sort by colour, shape and size and can eject defective and foreign pieces at up to a million objects a minute.

In the following years, Smalltalk proved to be “future safe” in two particular areas: longevity and innovation. Some of the software didn’t need to be changed; it works just as well on the modern sorters as it did with Tegra back in the mid-90s. On the other hand, new detection equipment could be integrated easily, especially the new advanced G6 platform, which became the base of all of Key Technology’s sorters such as Tegra, Optyx, Manta and VeriSym.

An optical sorter is operated not by computer specialists but by industrial production staff who have sometimes just been trained on the job. The machine operations and the wide range of configuration options based on complex sorting algorithms have to be presented by a simple and intuitive human-machine interface. All G6 machines are equipped with touch screens. There is no standard for such UIs. In addition, each type of use case, each new feature and each entry into a new market requires its own customised interface. Needless to say, the UI must be delivered in the operator’s spoken language. No standard operating system widgets can be used, and the UI has to be individually designed and developed.

Smalltalk allows Key to prototype and implement new sorting algorithms quickly so that more markets, including those outside the food industry, can be addressed. When Key found that the Windows operating system was too limiting, a switch to Linux was a no-brainer because of the superior cross-platform capabilities of VisualWorks. Even advanced needs could be addressed easily. For example, G6 machines can communicate with each other and with other machines using industry standard protocols, and operators can monitor and handle the machines via a secure remote interface from anywhere in the world.

Dedicated to quality, it is not surprising that Key Technology is an ISO-9001-certified company. Consequently, quality is also a central goal in their software development. One of the strengths of the Smalltalk development environment they see is the ability for the software team to constantly refactor its code base to improve it and to meet the ever-changing requirements of their customers. Key Technology has shipped hundreds of different sorter configurations and they know that if necessary, they can support tens of thousands. Quality, innovation and solutions with a focus on their customers’ individual needs make them the dominant player in their market. For whatever new requirements that come, Key feels confident in successfully addressing them with Smalltalk.

18.6 MetaCase—Software Development Tools

MetaCase, the Finnish pioneer of domain-specific modeling (DSM), provides tools and services for a more productive and higher quality approach to system and software engineering. The “meta” development environment MetaEdit+ lets developers use automatic code generation based on modeling languages and development processes that are specifically tailored to their domain. They enjoy the freedom to use what fits their special needs

without the straitjacket of fixed design methods and low-level languages and can create systems more efficiently and with fewer defects. MetaCase's customers such as Siemens, Denso, Panasonic and EADS use MetaEdit+ to improve productivity and outperform their competition in fields as diverse as railway systems, fish farming, insurance, home automation, telecom services and wearable sports computers.

MetaEdit+ consists of two tools: the Workbench and the Modeler. First, the Workbench is used to create the domain-specific language and process by defining the concepts, notation, rules and generators. Then the system engineers use this DSL in the Modeler with its editors, browsers and generators to actually build the system. MetaEdit+ offers advanced multi-user support. The common repository manages multi-user concurrency at a very fine-grained level allowing the developers to work on the same models, often even on the same objects, minimising lock-outs while ensuring there will be no design data conflicts. Without the need to compile a language, modifications to modeling languages and code generators are available to all users in real time. This way, complete modeling languages and models can be effectively shared and updated. MetaEdit+ is available on Windows, OS X and Linux and can be integrated into any modern software development tool chain including Eclipse and Visual Studio.

MetaCase required a powerful and flexible environment for the development and continuous enhancement of MetaEdit+, and Cincom VisualWorks was their tool of choice. Their decision to use Smalltalk was based on the product's dynamic technology with easy modeling and meta-programming capabilities and high developer productivity. The product's openness, robustness and platform independence were equally important.

MetaCase's roots began back in the late 1980s in the area of academia—where work is creative yet thorough and disciplined. Their research on situational development uncovered the required functionality for tools that could support individual, possibly not-yet-defined development processes, with MetaEdit (1991) being the first experimental implementation and MetaEdit+ (1995) the first industrial-scale product. MetaCase credits the feedback from its professional industrial users for taking MetaEdit+ from something “interesting” to something that is extremely useful. It is now known as a benchmark in the areas of professional usage and academic research and has won awards since its initial version.

18.7 MMA—*Insurance*

MMA (Mutuelles du Mans Assurances) is a mutual insurance group with almost 2,000 sales and customer services offices throughout France. The group, which was founded in 1828, has over three million clients consisting of individuals and professionals as well as businesses, communities and associations. With over 13,000 employees and 1,500 agents, MMA is the third largest agent network in France.

Back in the 1990s, MMA selected Smalltalk for the purpose of developing two business critical systems. These systems require high stability and excellent performance in order to support 11,300 simultaneous global users in back offices and front offices. Each day these

employees would be conducting business using 120 separate applications that had to be able to accurately access records for MMA's impressive customer base as well as its 690 products that were dispersed amongst 320 servers. To give an example of the load on the system, one particular monthly batch process handles 120,000 contracts in just one hour.

MMA's previous system had poor performance, and, using a rules-based engine, it wasn't possible to construct complex industry models. The new system that MMA built with Cincom VisualWorks is based on a framework with system parameterisation and offers an easy-to-use GUI. In addition, all interfaces are parameterised—including those that connect with various MMA internal groups and external agencies. The development team was able to lessen the code needed for upgrades and maintenance. In fact, as new or improved functions are requested, the new solution facilitates easy upgrades on an as-needed basis.

The two new solutions have been instrumental in helping MMA save time and run their business more efficiently. The first solution helps in performing sales-related tasks, which include the handling of contracts and the tracking and management of pre-sales, sales and invoicing processes. This solution runs in batch mode on the server, but also on the back-office client/server workstations and on a web client that is accessible by both the sales team and agencies. The second system, which requires telephony functionality, handles all claim-related processes. Both solutions integrate smoothly with other technologies such as Java, .Net, GraphTalk and COBOL. MMA programmers are happy that Smalltalk is easy to understand and at the same time offers powerful capabilities such as the ease of building a meta model for implementing complex industry models and application frameworks. They also appreciate the ability to reuse objects. All of these capabilities help developers become much more productive while working with a system that is enjoyable to use.

18.8 OOCL—*Logistics*

Orient Overseas Container Lines Ltd. (OOCL), headquartered in Hong Kong, China, is one of the leading international container transportation and logistics services providers. The firm serves clients within a wide geographical area including Asia, Europe, North America, the Mediterranean, the Indian sub-continent, the Middle East and Australia/New Zealand. OOCL is also an industry leader in using information technology and e-commerce for managing the entire cargo process.

OOCL is a global operation with more than 320 offices in 65 countries and over 9,200 full-time employees. Users from a wide variety of departments including financial accounting, customer service, vendor management, legal and sales access their online systems that keep track of 2.5 million annual shipments represented in 10 billion data objects. Since the information in the system can change quite rapidly and affect revenue opportunities and operation profitability, it requires minute-to-minute attention.

All aspects of OOCL's core business are coordinated by the "Integrated Regional Information System", which is known as IRIS-2. This includes initial order placement, customer service, moving goods and reconciling accounting. Thanks to IRIS-2, employees

from anywhere in the world have instant access to data. To make operations more efficient, container movements and costs must be tracked in real time. The ability to rapidly respond to customers is crucial for increasing revenue opportunities and outperforming competitors to win business.

To manage the processing load and keep data easily accessible and organised throughout its global organisational structure, OOCL moved to a distributed network of computing hardware. Rather than mapping information requests to database storage formats, they preferred to maintain the native object format for as much data as possible. Therefore, they chose GemStone/S—a Smalltalk object database and object server—to create a unique shipping management application. The technical capabilities and scalability of GemStone/S removed several mapping concerns, which resulted in extended programmer productivity. To allow all operations to be maintained in a Smalltalk object framework, they developed the front end using Cincom VisualWorks. This meant faster application development and integration and maximum performance for system end-users.

With IRIS-2, the backbone of the enterprise-wide information processing architecture, OOCL can deploy a wide range of user interface applications with real-time information. These tools such as CargoSmart, OperationSmart, DepotSmart and SchedulingSmart facilitate immediate customer, partner and vendor interaction. Increased operation automation and self-service opportunities require fewer employees in call centre and support operations. Also, since billing and tracking is much more coordinated, there has been a dramatic increase in fee collections for late customer pick-ups and container returns.

During the early 2000s, many shipping companies lost massive amounts of money due to the worldwide economic slowdown. However, OOCL was able to maintain and even raise its performance levels. In fact, since implementing IRIS-2, there has been a significant increase in employee efficiency, if measured by shipping tonnage per headcount.

18.9 Rudolph Technologies—Semiconductor Industry

Listed on the NYSE national market, Rudolph Technologies, Inc. is one of the global leaders in a highly specialised market that serves microelectronic device manufacturers. The company designs, develops, manufactures and supports systems and software for defect inspection, advanced packaging lithography, process control metrology and data analysis. Their portfolio is used in both the wafer processing and final manufacturing of integrated circuits (ICs) and in adjacent markets such as flat panel display, LED and solar. The proprietary products enable Rudolph's customers to drive down the costs and time-to-market of their own products.

Reduced time-to-market is indeed a key factor in the manufacturing industry, particularly in the competitive semiconductor capital equipment sector. Rapid prototyping is essential to market deployment, both from the perspectives of Rudolph software developers and their customers. Using Cincom VisualWorks, the firm created ControlWORKS® Software, an integrated development framework that provides 80 to 90 percent of the

functionality required to manage most semiconductor equipment. Customers can add to and customise semiconductor tools to create their own unique products. Using ControlWORKS, Rudolph customers can experience a drastic reduction in the time-to-market. One major customer's time-to-market was cut from 2 to 3 years to just 6 to 9 months while the control organisation was able to be reduced by up to 90 percent.

Rudolph often partners with its ControlWORKS customers. During such collaboration, the parties typically work together remotely and end up sharing code across the world in aid of multiple product development threads. ControlWORKS includes the whole toolset of VisualWorks, which teams of both Rudolph and its customers use as the development environment. Based on their experiences, Rudolph affirms that VisualWorks' code management tools support such collaborative distributed development very well. The Rudolph development team also appreciates the environment because it lets them easily try out new ideas and debug quickly and effectively in both development and run-time environments. In addition, it allows them to manage the code artefacts in a meaningful way.

Other big advantages of the pure, object-oriented Smalltalk environment include its increased flexibility, the reusability of code and the ease of code maintenance, which when combined, enables equipment makers to reduce the size of their software development team due to their increased productivity.

Today there are more than 15,000 ControlWORKS systems in use. ControlWORKS seamlessly integrates with Rudolph's advanced process control and yield management software, namely the Smalltalk-based run-to-run supervisory process control and optimisation system ProcessWORKS® Software.

18.10 SOOPS—*Energy Markets*

The founders of Soops, a Dutch software development company, are ardent supporters of object-oriented technology. They believe that it's the best choice for translating the ever changing needs and complex requirements of companies into functional ICT systems.

While concentrating on the financial services industry back in the 90s, they developed a stock trading system and soon realised that thanks to the system's generic character, other business sectors could benefit as well. This caused Soops to become a company that delivers transaction and order systems in real time and for virtually any market, one of which is the profitable area of energy exchanges, a market where Soops' solution became the de facto standard in Europe.

Soops' products are comprised of several different modules that work cohesively. Since each module resides on an individual server, scalability is solely a hardware issue. The team is able to add or remove products and markets without making major structural changes to the system thanks to the modular and generic architecture of their products.

Re-examining its Smalltalk technology choice against the latest technological advancements is part of a process that Soops does on a regular basis, and Cincom VisualWorks has always easily measured up. Due to high developer productivity, Soop's time-to-market far

undercuts those of its competitors. Thanks to Smalltalk's simplicity, business analysts are able to play an active role in spontaneous application development, which results in getting it right the first time. Also, since the entire Smalltalk code is open, vendor dependency is reduced, making it possible for Soops to take on its own problems. This helps in avoiding possible downtimes, which are detrimental to any exchange market.

Many years have passed, and Soops' applications have resulted in proven stability and scalability. Smalltalk has brought about the business agility that is essential in the marketplace today—for both Soops and its customers. Now, when market conditions change, it's easy for clients to quickly adjust their exchange systems and for Soops to penetrate emerging markets by adapting its generic solution.

Expanding the VisualWorks Image

After you install VisualWorks, the base Image (to be found in the subdirectory `image` in the installation directory¹) contains packages or bundles with the standard classes that are needed for application development, along with those that contain the program code for the development environment.

The subdirectory `contributed` contains several enhancements that are delivered with VisualWorks. Among these are

- Frameworks for the most diverse programming tasks (for example, SUnitToo, a later version of SUnit²)
- Components to access various database systems
- Free versions of third-part software
- Much more

These enhancements are in the form of *parcels*. Stated somewhat simply, parcels are packages that have been “exported” from an Image. You can use the menu item **Package → Publish as Parcel** to export a package as a parcel. This creates two files:

```
<package-name>.pst  
<package-name>.pcl
```

The file with the `.pst` extension contains the Smalltalk source code. The file with the `.pcl` extension contains the program translated into byte code.

You can use the **Parcel Manager**, which is started by clicking the menu item **System → Parcel Manager** in the Launcher, to integrate parcels into one’s own Image. The Parcel Manager presents three views of the parcels from the `contributed` directory.

¹See Fig. 5.1.

²See Chap. 15.

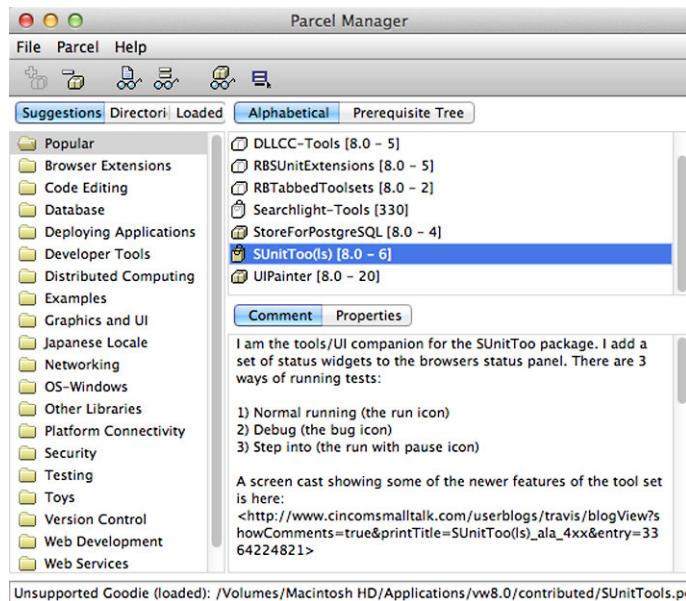


Fig. A.1 The Parcel Manager

1. The parcels are grouped by theme in the **Suggestions** tab. In Fig. A.1, the parcel SunitTool (s) has been selected from the Popular directory (see Chap. 15 and Appendix A.1).
2. In the left field of the **Directories** tab, the entire list of directories in the VisualWorks installation directory is shown (see Fig. 5.1).
3. From the **Loaded** tab, one can view which parcels have already been loaded.

A.1 Adding SunitToo

SUnit, described in Chap. 15, is already integrated into the VisualWorks base Image, and it can be used immediately. But Chap. 15 uses an alternative implementation of SUnit (SUnitToo), which can be integrated better into the System Browser. Although SUnitToo is also delivered with VisualWorks, it is not contained in the base Image. You must first load a parcel before you can use SUnitToo. To do that, the following steps are required:

1. Start the Parcel Manager with the Launcher menu item **System → Parcel Manager**.
2. In the Parcel Manager window, select the **Suggestions** tab.
3. Select the directory Popular and highlight the Parcel SUnitToo (ls) (see Fig. A.1).
4. Use the menu item **Parcel → Load** to load the parcel into the Image.

Fig. A.2 Connection information for the Cincom Public Repository



A.2 Adding the Object Explorer

Perform the following steps to load the package `ObjectExplorer`:

1. Connect your Image to the *Cincom Public Repository* (see Appendix A.4).
2. Use the Launcher menu item **Store → Published Items** to display a list of packages and bundles stored in the repository.
3. Select `ObjectExplorer` in this list.
4. Select the latest version on the right hand side of the window.
5. Use the menu item **Versions → load** to load the package into the Image.

A.3 Adding Seaside

You can find the Seaside framework in the Web Development directory in the **Suggestions** tab of the Parcel Manager. The parcel is called Seaside-All. The parcel Seaside-Examples-All contains a series of sample applications that were implemented using Seaside.

A.4 The Cincom Public Repository

If you have Internet access, you can connect your Image to the *Cincom Public Repository*. This is a Store repository,³ which is publicly accessible and in which you can find the most current versions of third-party software, which is to a large extent still in development.

In order to create the connection, from **Store** on the Launcher menu, select **Connect to Repository...** At that point, the window shown in Fig. A.2 appears. Be sure that Cincom Public Repository is selected as the *Connection Profile*. Do not change the connection

³See Chap. 17.

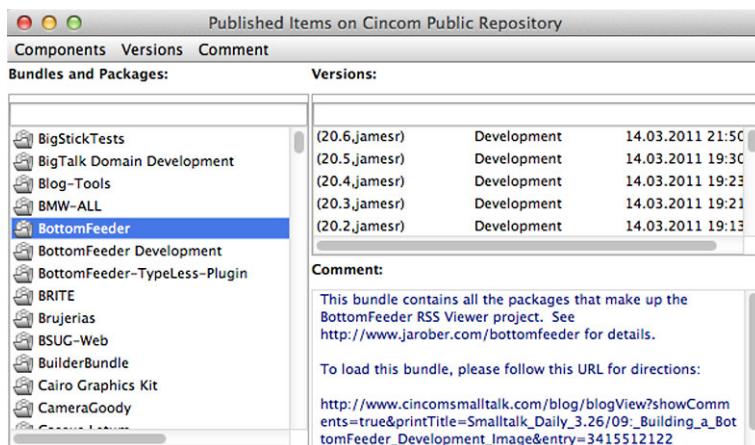


Fig. A.3 Published items in the Cincom Public Repository

information already on the screen. The connection will be established after you click **Connect**. Once the connection has been established, that fact will appear in the status bar of the Launcher.

Afterwards you can use the Launcher menu item **Store → Published Items** to display a list of packages and bundles stored in the repository. Figure A.3 shows part of the list; the bundle **BottomFeeder** has been selected, which contains a news reader. In order to load a package, highlight a version (in case of doubt, select the newest) and click the menu item **File → Load**.

References

- AIDAweb: AIDAweb Smalltalk Web Application Server (2013). <http://www.aidaweb.si>. Accessed 1 August 2013
- Alpert, S., Brown, K., Woolf, B.: The Design Patterns Smalltalk Companion. Software Patterns Series, vol. 1. Addison-Wesley, Reading (1998)
- Balzert, H.: Lehrbuch der Software-Technik, vol. 1, 2nd edn. Elsevier, Amsterdam (2001)
- Beck, K.: Smalltalk Best Practice Patterns. Prentice Hall, New York (1997)
- Beck, K.: Test-Driven Development by Example. Addison-Wesley, Reading (2003)
- Beck, K.: Simple Smalltalk testing: with patterns (2013). <http://www.xprogramming.com/testfram.htm>. Accessed 8 August 2013
- Ben-Menachem, M., Marliss, G.S.: Software Quality, Producing Practical and Consistent Software. Thomson Computer Press, London (1997)
- Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Conallen, J., Houston, K.A.: Object-Oriented Analysis and Design with Applications, 3rd edn. Addison-Wesley Professional, Reading (2007)
- Cincom Systems: VisualWorks Online Documentation. VisualWorks Walk Through. Version 7.8. Cincom Systems, Inc., Cincinnati (2011)
- Cincom Systems: VisualWorks Online Documentation. GUI Developer's Guide. Version 8.0. Cincom Systems, Inc., Cincinnati (2014a)
- Cincom Systems: VisualWorks Online Documentation. Application Developer's Guide. Version 8.0. Cincom Systems, Inc., Cincinnati (2014b)
- Cincom Systems: VisualWorks Online Documentation. Database Application Developer's Guide. Version 8.0. Cincom Systems, Inc., Cincinnati (2014c)
- Cincom Systems: VisualWorks Online Documentation. Source Code Management Guide. Version 8.0. Cincom Systems, Inc., Cincinnati (2014d)
- Cincom Systems: VisualWorks Online Documentation. Tool Guide. Version 8.0. Cincom Systems, Inc., Cincinnati (2014e)
- Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, San Diego (1972)
- Ducasse, S., Renggli, L., Shaffer, D.C., Zacccone, R.: Dynamic Web Development with Seaside. Square Bracket Associates, Kehrsatz (2010)
- Ernst, H.: Grundkurs Informatik, 4th edn. Vieweg+Teubner, Wiesbaden (2008)
- Fowler, M.: Refactoring – Wie Sie das Design vorhandener Software verbessern. Addison-Wesley, Reading (2000)

- Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
- Goldberg, A.: Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading (1983)
- Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison-Wesley, Reading (1989)
- Hopkins, T., Horan, B.: Smalltalk—An Introduction to Application Development Using Visual-Works. Prentice Hall, New York (1995). <http://stephane.ducasse.free.fr/FreeBooks>
- Jacko, J.A. (ed.): Human-Computer Interaction Handbook, 3rd edn. CRC Press, Boca Raton (2012)
- Link, J.: Unit Tests mit Java. Der Test-First-Ansatz. dpunkt.verlag, Heidelberg (2002)
- Mayr, H.C., Maas, J.: Perspektiven der Informatik. Informat.-Spektrum **25**(3), 177–186 (2002)
- Mössenböck, H.: Objektorientierte Programmierung in Oberon-2.2. Springer, Berlin (1994)
- Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd edn. Wiley, New York (2011)
- Oestereich, B.: Objektorientierte Softwareentwicklung: Analyse und Design mit der UML 2.0, 7th edn. Wissenschafts-verlag, Oldenbourg (2005)
- Pepper, P., Hofstede, P.: Funktionale Programmierung. Springer, Berlin (2006)
- Perscheid, M., Tibbe, D., Beck, M., Berger, S., Osburg, P., Eastman, J., Haupt, M., Hirschfeld, R.: An introduction to Seaside. Software Architecture Group. Hasso-Plattner-Institut (2008). <http://www.hpi.uni-potsdam.de/swa/seaside/tutorial>
- Preim, B.: Entwicklung interaktiver Systeme – Grundlagen, Fallbeispiele und innovative Anwendungsfelder. Springer, Berlin (1999)
- Rails: Web development that doesn't hurt (2013). <http://rubyonrails.org>. Accessed 25 August 2013
- Reiser, M., Wirth, N.: Programmieren in Oberon, Das neue PASCAL. Addison-Wesley, Reading (1994)
- Ritter, M.: CSS 4 you—die deutsche Seite für Cascading Style-Sheets (CSS) (2009). <http://www.css4you.de>. Accessed 29 August 2013
- Ruby: a programmers's best friend (2013). <http://www.ruby-lang.org/de/about/>. Accessed 25 August 2013
- Seaside: stairway to agile Web (2013). <http://www.seaside.st/>. Accessed 1 August 2013
- SUnit: the mother of all unit testing frameworks (2013). <http://sunit.sourceforge.net/>. Accessed 25 August 2013
- w3schools.com. <http://www.w3schools.com>. Accessed 8 October 2014
- Wirth, N.: Program development by step-by-step refinement. Commun. ACM **14**(4), 221–227 (1971)

Index

A

Absolute error, 156
Abstraction, 31
Acceptance test, 333
Aggregation, 307, 329
Algorithm, viii, 286
 iterative, 286
 recursive, 286
Architectural models, 402
Argument, 35
Array, 48
Assignment, 11, 49, 273, 280
Assignment semantics, 53
Association, 186, 307, 328
Attribute class, 186

B

Base, 150
Basic arithmetic functions, 152
Behaviour, 33
Binary method, 107
Binary symbols, 1, 8
Bit manipulation, 154
Block, 22, 40, 249
Block parameter, 251
Block variable, 75, 76
Bottom-up design, 321
Breakpoint, 209
Bundle, 88
Business logic, 307
Byte code, 9

C

Calculation time, 288
Call/Answer mechanism, 378
Callback, 363, 364, 372, 381
Calling method, 34
Calling pattern, 101
Cascade of messages, 84
Cascading Style Sheets, *see* CSS
Case-by-case distinction, 20
Character code, 167
Class, 68, 91, 93, 97, 144, 254, 264, 267, 269, 307
 abstract, 174
 Array, 48, 211, 226
 Association, 217
 Automaton, 258
 Bag, 216
 BlockClosure, 249
 Boolean, 163
 CcAdministration, 375, 377
 CcConverter, 350, 366, 367, 396
 Character, 48, 166
 Circle, 97
 Class, 271
 CompiledBlock, 250
 CompiledMethod, 250
 concrete, 174
 Controller, 323
 Converter, 346, 356, 394
 Currency, 346
 Date, 139, 171
 Dictionary, 217
 Double, 150, 156

-
- Class (*cont.*)
 ExternalReadStream, 305
 ExternalStream, 303
 ExternalWriteStream, 304
 False, 163
 Filename, 303
 Float, 46, 150, 156
 Fraction, 100, 148, 156
 Integer, 46, 146, 147, 149, 150
 InternalStream, 303
 Interval, 160, 232
 LargeNegativeInteger, 147
 LargePositiveInteger, 147
 LimitedPrecisionReal, 99, 150
 Magnitude, 175, 177
 Metaclass, 271
 Model, 323, 324
 MyCircle, 109
 Number, 100, 157
 Object, 98, 99, 144
 OrderedCollection, 227, 230
 Person, 231, 264, 277
 Point, 42
 PositionableStream, 302
 QuadrEquat, 180
 ReadStream, 299
 Rectangle, 41, 91
 ScheduledWindow, 44
 SequenceableCollection, 212
 Set, 211
 SmallInteger, 87, 147
 Solution, 181, 186
 State, 259
 Stream, 299
 String, 47, 168, 234
 Student, 265
 subclass, 98, 146, 324
 superclass, 98, 100, 119, 144, 146, 267, 324
 Symbol, 48, 234
 TextCollector, 83
 Time, 171, 173
 True, 163
 UndefinedObject, 50, 68
 View, 323
 WAComponent, 350, 373, 399
 WAFileLibrary, 392
 WATableReport, 399
 WriteStream, 299
 Class category, 89
 Class commentary, 91
 Class diagram, 324
 Class hierarchy, 97, 144, 269
 Class instance variables, 120
 Class library, 88, 144
 Class method, 41, 43, 93, 121, 311
 Class method protocol, 93
 Class relationships, 324
 Class variable, 52, 94, 139, 172
 Classes, 40
 Coercion, 162
 Collection run, 75
 Collections
 heterogeneous, 223
 homogeneous, 223
 ordered, 211
 unordered, 211
 Comments, 23
 Compiler, 9
 Component tests, 334
 Composed method, 28, 316, 317
 Computer, 1
 Configuration management, 403
 Container classes, 211
 Containment relationship, 329
 Context stack, 68
 Control variable, 75
 Correctness testing, 333
 Count loops, 73, 155
 CSS, 344, 355, 360, 361, 366, 386, 387, 389, 392
 CSS class, 387
 CSS ID, 387
- D**
- Data, 6
 Database access, 404
 Debugger, 68, 88
 Decision tree, 25
 Deep copy, 280
 Delegation, 184
 Dependency mechanism, 324
 Dependent objects, 323
 Design pattern, 402
 Development environment, 9
 Dialogue control, 343
 Dialogue window, 13
 Division, 153
 whole-number, 153

E

Embedding Seaside components, 400
Equality, 273
Error
 absolute, 156
 relative, 156
Exception handling, 195
Exceptions, 202
Execution sequence, 72
Exponential representation, 156

F

False, 52
File, 299
Fixed point, 134
Floating-point numbers, 149
Formal language, 4
Framework, 343
Function, 154
 mathematical, 154

G

Generality concept, 162
Generalization, 326
Get method, 104
Graphical User Interface, *see GUI*
GUI, 307, 321, 343

H

Hardware, 2
Hash function, 278
HTML, 343, 344, 389, 392, 399
 alignment of text in tables, 358
 boxes, 361
 buttons, 360, 361
 column headings, 357
 form, 361
 headings, 354
 input fields, 360, 361
 link anchor, 378
 rows, 357
 selection list, 370
 tables, 357
HTTP protocol, 353
HTTP request, 353, 363, 380

I

Identity, 273
Index, 65
Information, 4

Information hiding, 33

Inheritance, 97, 100, 120, 143, 264, 307, 324

Initialization, 63

Inspector, 41, 86

Instance method, 93, 124, 308

Instance variable, 34, 42, 52, 86, 88, 91, 98,
 103, 104, 106, 120, 180, 200, 205, 210,
 308, 329

Integer class, 150

Interface, 33

Interpreter, 9

Interval, 74, 159

Interval run, 74

Isa relationship, 326

Iteration, 64

J

JavaScript, 400

K

Keyword message, 36, 37
Keyword method, 105, 106

L

Launcher, 80
Literals, 41, 45, 150
Loop, 63

M

Memory, 289
Message
 binary, 36
 keyword, 36, 37
 unary, 36
Message expression, 36
Message pattern, 101
Message selector, 35, 101
Metaclass, 144, 269
Metaprogramming, 270
Method, 33, 41
 abstract, 176
 binary, 107
 class, 41, 43, 93, 311
 generic, 176, 177
 instance, 93, 124, 308
Method activation, 34
Method body, 102
Method implementation, 93
Method protocol, 41, 90, 126, 307
Method search, 264
Micropattern, 402

-
- Model, 31, 346
 Model building, 31
 Model-View Controller paradigm, 321, 344, 402
 Modelling, 31
 Multiple inheritance, 326
- N**
- Namespace, 115
 Nil, 52
- O**
- Object
 - behaviour, 33
 - status, 33
 Object copies, 279
 Object equality, 277
 Object sharing, 274
 Object-oriented analysis, 130, 401
 Object-oriented design, 130, 401
 Outer context, 251
- P**
- Package, 88, 115, 404
 Parameter, 39
 - formal, 106, 121
 Pattern matching, 171
 Polymorphism, 143, 180
 Program, 4
 Programmability, 2
 Programming languages, 3
 Pseudo-variable, 36, 42, 70, 102, 163, 267
- R**
- Recursion depth, 293
 Refactoring, 236, 317
 Reference semantics, 53
 Reflection, 270
 Regression test, 317, 334
 Relative error, 156
 Repetition, 59, 63
 Return instruction, 103
 Return operator, 103, 122
 Reuse, 98
 Ruby on Rails, 344
- S**
- Scientific notation, 156
 Scope of validity, 106, 200
 Seaside, 344, 349, 399
- Seaside application, 352, 353
 Seaside callback, 363, 372, 373, 381
 Seaside component, 350
 Seaside Dispatcher, 352
 Seaside error message, 368
 Seaside file library, 392
 Seaside Halos, 354
 Seaside messages, 359
 Seaside server, 352
 Seaside Sessions, 400
 Seaside standard dialogue, 373, 382, 384
 Self, 52, 70, 102, 103, 106, 107, 122, 137, 264, 265, 267
 Semantics, 4
 Set method, 104, 124
 Shallow copy, 280
 Side effect, 55, 275
 Software, 2
 Software engineering, 2, 113, 401
 Software ergonomics, 402
 Specialization, 326
 Specification, 61
 State, 33
 State of execution, 72
 Step-by-step refinement, 28, 61, 315, 320
 Stream, 299
 - external, 303
 - internal, 303
 Structured programming, 315
 Stylized text, 64
 Subscript, 65
 SUnit, 335
 Super, 52, 137, 264, 266, 267
 Syntax, 4
 Syntax errors, 199
 System browser, 88
 System test, 333
- T**
- Termination, 63, 295
 Test-driven development, 341
 Test-first approach, 341
 TestCase, 335
 TestResult, 335
 TestSuite, 335
 ThisContext, 52
 Top-down design, 241, 320
 Transcript, 80
 Translator, 9
 True, 52

U

UML, 185
Unified Modelling Language, 185
Unit test, 333
User interface, 307, 402

V

Value semantics, 53
Variable, 49, 51
 assignment, 280
 class, 52, 94, 139, 172
 declaring, 10, 102
 global, 52
 instance, 34, 42, 52, 86, 88, 91, 98, 103, 104,
 106, 120, 180, 200, 205, 210, 308, 329
 internal, 33
 local, 10
 private, 52

temporary, 10

Variable declaration, 102
Variable identifier, 52
Virtual machine, *see* VM
VM, 8

W

Watchpoint, 210
Web application, 343
Wild cards, 170
Workspace, 10, 85
Workspace variable, 83, 85

X

XHTML, 344

Y

Yourself, 223, 228