Dave’s Smalltalk FAQ

Email: dnsmith@watson.ibm.com

Last-modified: 28 July 1995

Certain questions get asked time and again. In order to save the repeated answering of these questions, and to provide the best answers, collections of these ’frequently asked questions’, and their answers, of course, are made and distributed to interested parties. Called FAQs, these documents often become the basic documentation on a topic and should be checked before posting a question to a related newsgroup.

This file is a set of frequently asked questions for the Smalltalk language. There are several such Smalltalk FAQs; this one is by David N. Smith.

Thanks to all those on comp.lang.smalltalk who asked questions, answered questions, and provided an inspirational environment in which the seeds of this FAQ grew and propsered.

Format and Availability

A table of contents to this FAQ is distributed periodically on the newsgroup `comp.lang.smalltalk`. It consists of a text version of part one of the FAQ.

The FAQ itself is available in several formats including Postscript™ and Adobe™ Acrobat™ formats. It is available online at:

URL: `http://st-www.cs.uiuc.edu/users/dnsmith/SmallFaq.html`

and

URL: `http://www.dnsmith.com/SmallFAQ/SmallFaq.html`

There is no text-only version; not providing one is somewhat unusual, but so is having a typeset FAQ. The fonts used are the ever-present Times and Courier, selected because every postscript printer known to man has them builtin. It is thus not necessary to distribute copies of the fonts in the postscript and Acrobat files.At the moment there is no HTML version; one will appear when suitable and satisfactory methods of translating Adobe FrameMaker files to HTML are available.

If you have not already read the overall Usenet introductory material posted to "news.announce.newusers", please do. It is also available by ftp in:

URL: `ftp://garbo.uwasa.fi/pc/doc-net/usenews.zip`

Status

This is a draft document. It surely contain errors in its statements and bugs in its code. Some obvious questions are not present. Many questions are asked but left unanswered. The obvious questions will be asked and the unanswered questions will be answered in the fullness of time. Chapters nearer the front of the document are more complete than chapters nearer the back.

There are many notes from the author to the author:

These are usually right aligned and flagged with a triangle.◄

Craig Latta's Smalltalk FAQ

Another Smalltalk FAQ is by Craig Latta and is available in text format at:

URL: `ftp://XCF.Berkeley.EDU/pub/misc/smalltalk/FAQ/FAQ.txt`

or:

URL: `ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/comp/lang/smalltalk/`

and in HTML format at:

URL: `http://XCF.Berkeley.EDU/pub/misc/smalltalk/FAQ.`

Vikas Malik's Smalltalk FAQ

Yet another Smalltalk FAQ is by Vikas Malik and is available in HTML format at:

URL: `http://www.infi.net/~vmalik/`

The Object People's IBM Smalltalk FAQ

The Object People maintain a FAQ on IBM Smalltalk at:

URL: `http://www.objectpeople.on.ca/~wayne/faq/visualage/`

IBM’s Smalltalk FAQ

IBM maintains a FAQ on IBM Smalltalk at:

URL: `http://www.torolab.ibm.com/software/ad/vishints.html`

Trademarks

Many trademarks are used in this FAQ. These trademarks are the property of their respective owners. When known to the author, trademarks are indicated by an uppercase leading letter, or a word entirely in uppercase letters as appropriate.

Quoted material without other attribution is from an obvious (and referenced) URL.

Updates and Corrections

Comments from readers are not only welcome but are invited.

- Are the questions the right ones?
- Are the answers suitable and at the right level?
- What topics need to be covered?
- What questions should be added?

The author does not have access to all existing Smalltalk system (nor the time to be an expert in all the ones he does have). If you have information on specific implementations that should go into any question, please forward it to the author.

If you have corrections or suggestions for this FAQ, send them to David N. Smith at `dnsmith@watson.ibm.com` or `dnsmith@dnsmith.com`.

Please include: your name, your email address; and your telephone number or snail-mail address. Submission of an update, correction, or additional topic grants permission to use the information in the FAQ and derivitive works if any, unless otherwise stated. Acknowledgement of sources will be limited to the submitters name and email address, unless requested otherwise, and subject to the editors judgement.

Thank you.

Short Table of Contents

DRAFT

Full Table of Contents

The Smalltalk Language

Magnitudes

DRAFT

Part I: General Questions

Frequently Asked Question

1.1 • What companies sell Smalltalk implementations?

Smalltalk is sold by VMARK, IBM, ParcPlace-Digitalk, GemStone and QKS Systems.

See “Smalltalk Implementations” on page 21 for detailed information.

Frequently Asked Question

1.2 • Are there any free Smalltalk implementations?

There are several free versions of Smalltalk: GNU Smalltalk, Tiny Smalltalk, SELF-Smalltalk, and Smalltalk-X.

See “Smalltalk Implementations” on page 21 for detailed information.

Frequently Asked Question

1.3 • Where can I buy Smalltalk Stuff?

Smalltalk stuff can be purchased from Smalltalk system and addon vendors, as well as from various programmers stores and catalogs.

See “Smalltalk Implementations” on page 21 for detailed information on vendors and third-party addons.

The Smalltalk Store, Silicon Valley

URL:	`http://www.smalltalk.com`
EMail:	Internet: `doug@smalltalk.com` CompuServe: `75046,3160` Telnet: `bbs.smalltalk.com` BBS: `1-415-854-5881`
Address:	The Smalltalk Store, 405 El Camino Real, Box 106, Menlo Park, CA 94025, USA
Phone:	1-415-854-5535
FAX:	1-415-854-2557

Douglas C. Shaker is the owner. They carry all things Smalltalkish.

Suggestions welcome: Where do *you* buy Smalltalk stuff?

Frequently Asked Question

1.4 • Is there a Smalltalk language standard?

No, but one is being developed.

See “ANSI Smalltalk” on page 22 for further details.

Learning the Smalltalk Language

Frequently Asked Question

1.5 • What is the best way to learn Smalltalk

(Suggestions anyone?)

Frequently Asked Question

1.6 • What Universities have Smalltalk courses?

Carleton University

Contact:	Professors John Pugh or Wilf LaLonde.
Address:	Ottawa, Ontario, Canada

Ryerson Polytechnic University

Contact:	Joshua Panar or Paul A. Salvini
EMail:	`psalvini@acs.ryerson.ca`
Address:	Department of Math, Physics and Computer Science Ryerson Polytechnic University Room A-100, Jorgenson Hall 380 Victoria St., Toronto, ON M5B 2K3

University of Illinois at Urbana-Champaign.

Contact:	Professor Ralph Johnson.

1.7 • What general books are available on the Smalltalk Language?

- *Concepts of Object Oriented Programming*
  Smith, David N.; McGraw-Hill, 1991, US$25. ISBN 0-07-059177-6. 208 pages.

  Teaches the concepts of object-oriented programming using Smalltalk; vendor independent. German edition due in 1996.

- *Developing Visual Programming Applications Using Smalltalk*
  Linderman, Michael; SIGS Books, 1996; SIGS ISBN 0-884842-28-3; Prentice-Hall ISBN 0-13-569229-6.

  Introduction to VisualAge, VisualSmalltalk, and VisualWorks with the emphasis in visual construction ot applications in Smalltalk.

- *Rapid Software Development with Smalltalk*
  Lorenz, Mark; SIGS Books, 1995, US$24. ISBN 1-884842-12-7 (SIGS). ISBN 0-13-449737-6 (Prentice-Hall). 238 pages.

  How to develop rapidly using Smalltalk. This is neither a language book nor a design methodology book, but a how-to book for Smalltalk programmers.

- *Smalltalk with Style*
  Skublics, Suzanne, Edward J. Klimas, and David A. Thomas, Prentice-Hall, 1996, US$15. ISBN 0-13-165549-3. 142 pages.

  URL: `http://www.prenhall.com/~rich/013/165548/16554-8.html`

  A vendor-independent style guide for Smalltalk. This is a book that all Smalltalk programmers should have; it tells beginners a lot about how to code the way experienced Smalltalkers code, and reminds experienced programmers of what they should be doing(!).

1.8 • What books cover the original Smalltalk-80?

- *Smalltalk-80: The Language and Its Implementation*
  Goldberg, Adele & David Robson; Addison-Wesley, 1983 (reprinted 1985 with corrections). ISBN 0-201-11371-6.

  This is the so-called *Blue Book*, because of its front cover color. It is out of print, but can be found used.† Contains an example implementation of Smalltalk

†. Powell’s often has used copie; see [1.16] *’Where can I buy Smalltalk books?’* on page 10.

written in Smalltalk -- obviously not for running, but for learning, it's nice. While the class library described was incomplete when published and is very out of date, the book is still referenced.

See *1.16 'Where can I buy Smalltalk books?'* Some stores carry used computer books.

- *Smalltalk-80: The Language*
  Goldberg, Adele & David Robson; Addison-Wesley, 1989. ISBN 0-201-13688-0. 608 pages.

  This is the front half of *Smalltalk-80: The Language and Its Implementation*. It is sometimes called the *Purple Book*, due to its front cover color. It is in print.

- *Smalltalk-80: Bits of History and Words of Advice*
  Krasner, Glenn; Addison-Wesley, 1983. ISBN 0-201-11669-3. 350 pages.

  A collections of early papers on Smalltalk implementation and on the history of Smalltalk. It is sometimes called the *Green Book*, due to its front cover color. For those interested in the history of Smalltalk and the otherwise terminally curious.

- *Smalltalk-80: The Interactive Programming Environment*
  Goldberg, Adele; Addison-Wesley, 1984. ISBN 0-201-11372-4. 528 pages.

  How to use Smalltalk-80. Describes general interface use, the text editor, projects, evaluating expressions, inspectors, browsers, creating classes, debuggers, and more. It is sometimes called the *Orange Book*, due to its front cover color. Quite obsolete. For those interested in the history of Smalltalk and the otherwise terminally curious.

Frequently Asked Question

1.9 • What books are available on ParcPlace Smalltalk-80?

- *Inside Smalltalk, Volume I*
  LaLonde, Wilf R.; and John R. Pugh, Prentice-Hall, 1990. ISBN 0-13-465964-3. 528 pages.

  The first volume of two. It covers fundamentals, programming and debugging, the core classes including magnitudes and collections, and graphics. Unfortunately, the version of Smalltalk described is no longer available.

- *Inside Smalltalk, Volume II*
  LaLonde, Wilf R.; and John R. Pugh, Prentice-Hall, 1991. ISBN 0-13-468414-1. 576 pages.

  URL: `http://www.prenhall.com/~rich/013/465963/46596-3.html`

  The second volume of two. It covers windowing and user interface issues. Unfortunately, the version of Smalltalk described is no longer available.

- *Object-Oriented Engineering: Building Engineering Systems Using Smalltalk-80*

Bourne, John R.; Asken Associates, R D Irwin Inc., 1992. ISBN 0-256-11210-X.

This is a textbook which “focuses on the understanding and use of object-oriented methodologies for engineering problem solving with a specific emphasis on analysis and design”.

Frequently Asked Question

1.10 • What books are available on ParcPlace VisualWorks?

- *Art and Science of Smalltalk, The*
  Simon Lewis. Prentice-Hall, 1995. 238 pages.

  URL: `http://www-uk.hpl.hp.com/people/scrl/ArtAndScience/home.html`
  `http://www.prenhall.com/013/371344/37134-4.html`

  An introduction to Smalltalk using VisualWorks. Covers the language, using the environment, the library, collections, dependencies, model-view-controller (MVC), pluggability and adaptors, designing and coding, debugging, and managing projects.

- *Smalltalk: An Introduction to Application Development Using VisualWorks*
  Hopkins, Trevor, Bernard Horan; Prentice-Hall, 1995, US$34. 426 pages. ISBN 0-13-318387-4.

  “This book is a complete, stand-alone introduction to application development using Smalltalk-80. It provides a comprehensive description of the VisualWorks 2.0 development environment, the language and major aspects of the class library, including coverage of the Model-View-Controller (MVC) paradigm. This book is aimed at students attending university/college and software professionals in industry. ... [N]o previous exposure to VisualWorks, Smalltalk-80 or object-oriented programming is assumed.”

  URL: `http://www.prenhall.com/013/318386/31838-6.html`

- *Smalltalk Developer’s Guide to VisualWorks, The*
  Howard, Tim; SIGS Books, 1995. ISBN 1-884842-11-9 (SIGS) and ISBN 0-13-442526-X (Prentice-Hall). 646 pages.

  An advanced level book covering the building of interactive applications with VisualWorks. Starts off with a description of Model-View-Controller (MVC), one of the few such descriptions in the literature.

- *Smalltalk Developer's Guide, With CDROM*
  Pletzke; MacMillan Computer Pub. ISBN 0-67-230720-0

  Not Yet Published. No further information. May not be VisualWorks book.

1.11 • What books are available on IBM Smalltalk?

- *AS/400 Application Development with VisualAge For Smalltalk (Bk/Disk), 1/e*
  Andreas Bitterer, Gino Porciello, John Oosthuizen, Masahiko Hamada,and Hakon Rambek; Prentice-Hall, June 1996, 416 pp.; Paper Bound with Disk (0-13-520453-4).

  "Covers the steps of analysis, design, and implementation of a typical application. Explains how to use DB2/400 through APPC routers and through Open Database Connection (ODBC).."

- *IBM Smalltalk Programming for Windows and OS/2*
  Shafer, Dan; Prima, 1995, US$49.95. 496 pages.

  Develops several live Smalltalk applications; source code diskette.

- *IBM Smalltalk: The Language*
  Smith, David N.; Addison-Wesley (Benjamin/Cummings), 1995, about US$49. ISBN 0-8053-0908-X. 600 pages.

  URL: `http://aw.com/bc/authors/smith/smalltalk/smalltalk.html`

  A very detailed coverage of the IBM Smalltalk language, core libraries, processes, exceptions, files, and related material. Little coverage of widgets or graphics.

- *IBM Smalltalk: Applications and Interfaces*
  Smith, David N.; Addison-Wesley, 1997. Approximately. 750 pages.

  Not yet published. It is claimed this book will have a very detailed coverage of building user interfaces, the graphics and widgets classes in IBM Smalltalk, and how to build applications.

- *VisualAge for Smalltalk Distributed: Developing Distributed Object Applications*
  Walter Fang, Sven Guyet, Matti Vilmi, and Eduardo Eckmann; June 1996, 288 pp., Paper (0-13-570805-2).

  "This book contains an overview of the features and architecture of Smalltalk's Distributed feature; sample application components with supporting documentation to illustrate design and coding; and recommendations for building distributed object applications with VisualAge."

- *VisualAge for Smalltalk SOMsupport: Developing Distributed Object Applications*
  Walter Fang, Raymond Chu, and Markus Weyerhauser; July 1996, 240 pp., ISBN 0-13-570813-3.

  "Provides guidelines for applying IBM's Visual Modeling Technique (VMT) to application modeling, design and implementation in a distributed object environment. Walks through the planning, analysis, design and implementation of a sample application. Explains how IBM's System Object Model (SOM) implements the industry-wide CORBA standard."

Frequently Asked Question

1.12 • What books are available on Digitalk Smalltalk?

- *Discovering Smalltalk*
  LaLonde, Wilf; Benjamin/Cummings, 1994. ISBN 0-8053-2720-7. 576 pages.
  An introductory textbook on Smalltalk. Can be used as an introduction to programming or just as an introduction to Smalltalk. The author is a professor at Carleton University and the book was developed for use there.
- *Smalltalk Programming for Windows*
  Shafer, Dan; Prima, 1993; US$39.95. ISBN 1-55958-237-5. 400 pages.
  Develops several live Smalltalk applications; source code diskette.
- *Smalltalk V: Practice and Experience*
  LaLonde, Wilf, John Pugh; Prentice-Hall, 1994, 192 pages. ISBN 0-13-814039-1.
  URL: `http://www.prenhall.com/~rich/013/814038/81403-8.html`
  A collection of the authors columns from the *Smalltalk Report*, this book contains lots of examples and comes with a diskette. Topics include generalization, disk objects, animation, windows and events, panes, modal components, fuzzy sets, and dynamic data exchange.
- *Object-Oriented Programming With Smalltalk/V*
  Marchesi, Michele; Prentice-Hall, 1993, US$36. ISBN 0-13-630294-7
  An introduction to OOP, using Smalltalk/V-286. Introduces OOP concepts, the Smalltalk/V language and core libraries. Includes three sample applications.

Frequently Asked Question

1.13 • What other Smalltalk books are there?

- *A Little Smalltalk*
  Budd, Timothy; Addison-Wesley, 1987. 296 pages. ISBN 0-201-10698-1.
  Little Smalltalk is a, well, very little implementation of Smalltalk that has no graphical user interface. Implementations exist for a number of platforms and source is available. The book describes both the internals and the externals of the

implementation.

See“Little Smalltalk” in section "Little Smalltalk" on page 27.

- *Design and Evaluation of a High Performance Smalltalk System, The*
  Ungar, David Michael; MIT Press, 1987. ISBN 0-262-21010-X. 288 pages.

  This book was an ACM Distinguished Dissertation in 1986. It describes a computer processor, the SOAR (Smalltalk on a RISC), which was designed by Ungar and others at Berkeley for the purpose of running Smalltalk. The book contains much information on Smalltalk software performance issues, and describes Ungar’s generation scavenging garbage collector.

- *History of Programming Languages*
  Bergin, Thomas J, and Ricnard G. Gibson, eds.; ACM Press/Addison-Wesley, 1996. 864 pages. US$49.50. ISBN 0-201-89502-1.

  Papers from the second History of Programming Languages conference in 1993, plus transcripts of talks, visual materials, and more. The history if Smalltalk is told by Alan Kay, pages 511-596. Other languages include: Algol 68, Pascal, Ada, Lisp, Prolog, Forth, C, C++ and some more obscure languages.

Frequently Asked Question

1.14 • What related books might be of interested to Smalltalkers?

- *Designing Object-oriented User Interfaces*
  Collins, Dave; Benjamin/Cummings, 1995, US$50. 608 pages.

  How to build object-oriented user interfaces from both the cognitive psychology and the programmers perspectives. Examples in Smalltalk (and C++).

- *Object-Oriented Application Frameworks*
  Lewis, Ted; and others. Manning Publications, 1995, US$30. 350 pages.

  Containing chapters by many experts in the frameworks fields including Erich Gamma, Andre Weigand, Larry Rosenstein, Glenn Andert, John Vlissides, Paul Calder, Kurt Schmucker, and Wolfgang Pree, it covers many successful application frameworks including *MacApp*, the Microsoft foundation classes, *ET++*, *InterViews*, Taligent object frameworks, *UniDraw*, and *ProGraph CPX*. Unfortunately, no Smalltalk frameworks are documented.

- *Pitfalls of Object-Oriented Development*
  Webster, Bruce F.; M&T Books, 1995, US$24.95. 272 pages.

  While this is not strictly a Smalltalk book, it covers many of the things that can go wrong when applying Smalltalk (or other OOP technology).

1.15 • What pattern books relate to Smalltalk?

While there are yet no books on patterns specifically for Smalltalk, the following books have some Smalltalk coverage.

- *Design Patterns*
  Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley, 1995. 416 pages.

  URL: `http://aw.com/cp/Gamma.html`

  This book "shows the role that patterns can play in architecting complex systems" and "it provides a very pragmatic reference to a set of well-engineered patterns that the practicing developer can apply". (From the forward by Grady Booch.) The examples are predominantly in C++ with a few in Smalltalk.

- *Pattern Languages of Program Design*
  Coplien, James O., and Douglas C. Schmidt, editors. Addison-Wesley, 1995, US$36.95. 576 pages.

  This is the proceedings of the first conference on Pattern Languages of Program Design (PLoP) in 1994. There are Smalltalk patterns in a number of papers.

- *Pattern Languages of Program Design 2*
  Vlissides, John M., James O. Coplien, and Norman L. Kerth, editors. Addison-Wesley, 1996, US$36.95. 634 pages. ISBN 0-201-89527-7.

  This is the proceedings of the second conference on Pattern Languages of Program Design (PLoP) in 1995.

- *Pattern-Oriented Software Architecture: A System of Patterns*
  Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal; Wiley and Sons Ltd., May 1996. ISBN: 0-471-95869-7. $39.95 US

  "The 25 patterns that are presented in this book span several levels of abstractions from high-level architectural frameworks and medium-level design patterns to low-level idioms. The book offers a new way of thinking about software architecture and teaches a number of techniques for solving particular recurring design problems."

1.16 • Where can I buy Smalltalk books?

After you’ve tried your local book dealer (and pointedly asked them if they carry the book, and why not!), check on Yahoo (or your favorite WWW index) for technical book stores in your area. Or try Ecola’s for a nearby technical book store:

List of Technical Book Stores on the WWW

URL: `http://www.ecola.com/ez/books.htm`

Ecola's Technical Book Spot has a list of technical book stores arranged geographically.

Some technical book stores have good mail order departments. The first two below are ones that I have used successfully and happily.

Also see: “The Smalltalk Store, Silicon Valley” in "1.3 Where can I buy Smalltalk Stuff?".

Computer Literacy Book Stores, Silicon Valley (3 stores), plus Washington, DC area (one store).

URL: `http://www.clbooks.com/`

New technical books. Mail order. Huge stock. They have a WWW-accessable search engine of what's in stock plus ordering instructions, phone numbers, etc.

Powell's Technical Books, Portland, Oregon.

URL: `http://www.technical.powells.portland.or.us/`

New and used technical books. Mail order. Huge stock. They have a WWW-accessable search engine of what's in stock plus ordering instructions, phone numbers, etc.

The stores below have a good net presence.

Lib HiTech

URL: `http://www.libhitech.com/libhitech/`

LibHiTech sells computer books, and appears to operate exclusively on the WWW.

ProTech Books, Ltd.

URL: `http://www.pro-tech.com/`

ProTech Books, Ltd., has three technical/ professional bookstores in the Dallas/ Ft. Worth metroplex, and a new one in Austin. The technical stores formerly did business as Taylor’s Technical Books. They sell computer, business, science, engineering, and junior tech books.

San Diego Technical Books, Inc.

URL: `http://www.sdtb.com/main.html`

San Diego Technical Books, Inc. is in San Diego county.

Conferences

Frequently Asked Question

1.17 • What is the ECOOP conference series?

URL: `http://iamwww.unibe.ch/ECOOP/`

The ECOOP Conference series is held annually in the summer in Europe. “The conference aims at bringing together researchers and practitioners from academia and industry to discuss and exchange new developments in object-oriented languages, systems and methods.”

ECOOP '96

URL: `http://www.ifs.uni-linz.ac.at/ecoop96`

The 10th European Conference on Object-Oriented Programming will be held in Linz, Austria, 8-12 July 1996.

Frequently Asked Question

1.18 • What is the OOPSLA conference series?

URL: `http://info.acm.org/sig_forums/sigplan/oopsla/oopsla.html`

The OOPSLA conferences are held annually in October in the United States or Canada.

“The annual OOPSLA (Object-Oriented Programming Systems, Languages and Applications) conference is the premiere forum bringing together participants in the field of object technology to share ideas and experiences. The combination of research and experience papers attracts both academic and industrial participants and ensures a rich exchange of ideas.”

"OOSPLA is the oldest and most prestigious object conference. OOPSLA is a unique experience that blends technology research, application development and practical trends and methods. OOSPLA is an opportunity for sharing, growing, learning, showcasing technology and exchanging ideas."

OOPSLA '96

OOPSLA '96 will be held 6-10 October 1996 in San Jose, California, USA.

URL: `http://www.acm.org/sig_forums/sigplan/oopsla/oopsla96/oopsla96.html`

OOPSLA '97

OOPSLA '97 will be held 5-9 October 1997 in Atlanta, Georgia, USA.

URL: `http://www.acm.org/sig_forums/sigplan/oopsla/oopsla97/oopsla97.html`

Table of contents of all OOPSLA Proceedings

URL: `http://sunsite.informatik.rwth-aachen.de/dblp/db/conf/oopsla/index.html`

Frequently Asked Question

1.19 • What are the SIGS conferences?

SIGS Conferences include Object Expo, Object Expo Europe, C++ World, and an annual Smalltalk conference held in late winter in New York City.

URL: `http://www.sigs.com/index.html`

Object Expo

August 5-9, 1996, New York Hilton & Towers, New York, NY

Object Expo Europe

September 23-27, 1996, QEII Conference Centre, London, UK

C++ World

November 11-15, 1996, Grand Kempinski Hotel

Object Expo Switzerland

November 18-20, 1996, Zurich

Object Expo France

December 2-6, 1996, Le Palais des Congres de Paris, Paris, France

Frequently Asked Question

1.20 • What is the Object World conference series?

URL: `http://www.omg.org/ow/objwrl.htm`

“Object World is the largest all object technology (OT) series in the world, with shows in Boston, Frankfurt, London, San Francisco, Sydney, and Tokyo. Object World focuses specifically on the commercial and practical aspects of applying object technology.”

Object World West ’96

August 18-22, 1996, San Jose Convention Center, San Jose, CA, USA.

Object World Australia '96

August 27-30, AJC Convention and Exhibition Centre, Sydney Australia

Object World Tokyo ’96

October 2-4.

Object World Frankfurt '96

October 9-11, Sheraton Conference Center, Frankfurt/Main, Germany.

Smalltalk Places on the Net

Frequently Asked Question

1.21 • Where are the Smalltalk code archives?

In addition to the web-based archives listed below, there is a sizeable archive of Smalltalk code in the Digitalk Forum (GO DIGITALK) on the CompuServe Information Service.

Archive of comp.lang.smalltalk

URL: `gopher://vm.gmd.de/11/listserv/Logs for Listserv Lists/INFO-CLS`

All of the appends from the newsgroup `comp.lang.smalltalk` are archived here.

Australian Smalltalk Web Server

URL: `http://www.smalltalk.org.au/index.html`

This Web server is a result of a joint effort between Gunn Software Pty. Ltd. and FreeFall Software Pty Ltd., both from Sydney, Australia.

Manchester Smalltalk Archive

The granddaddy of the Smalltalk archives, this archive was started by Mario Wolczko and Alan Wills. While it is no longer open for business under its own name, its contents are a part of the archives at the University of Illinois and Washington University St. Louis.

Smalltalk Archive at University of Illinois

URL: `http://st-www.cs.uiuc.edu/`

The Smalltalk Archive at the University of Illinois, Urbana-Champaign, is run by Ralph Johnson and his students. In addition to its own material, it contains the contents of the Manchester Smalltalk Archive.

URL: `ftp://ftp.cs.uni-magdeburg.de/pub/Smalltalk/`

A mirror of the UIUC Smalltalk archive.

Smalltalk Archive at Washington University St. Louis

URL: `http://wuarchive.wustl.edu/languages/smalltalk/`

The Washington University St. Louis archive mirrors the UIUC archives, Manchester archives, papers, patterns information, and more.

Frequently Asked Question

1.22 • What Smalltalk books have a net presence?

See the entries for the individual books; if a book has a net presence it's URL is shown with its description.

Frequently Asked Question

1.23 • What information is there about patterns?

“Patterns are the recurring solutions to the problems of design. People learn patterns by seeing them and recall them when need be without a lot of effort. Patterns link together in the mind so that one pattern leads to another and another until familiar problems are solved. That is, patterns form languages, not unlike natural languages, within which the human mind can assemble correct and infinitely varied statements from a small number of elements.”

—The Portland Pattern Repository

Christopher Alexander: An Introduction for Object-Oriented Designers

URL: `http://g.oswego.edu/dl/ca/ca/ca.html`

"In part because it is considered as much artistry as engineering, writings about architecture have most extensively explored and argued out the basic underpinnings of design. Even within this context, the ideas of the architect Christopher Alexander stand out as penetrating, and bear compelling implications for software design. Alexander is increasingly well-known in object-oriented (OO) design circles for his influential work on 'patterns'. This review introduces some highlights of Alexander's work."

Design Patterns Mailing Lists by thread

URL: `http://iamwww.unibe.ch:80/~fcglib/WWW/OnlineDoku/archive/DesignPatterns/`

The design patterns mailing list archive, by thread.

History Of Patterns

URL: `http://c2.com/cgi-bin/wiki?HistoryOfPatterns`

How Ward and Kent started it all...

Patterns Home Page (UIUC)

URL: `http://st-www.cs.uiuc.edu/users/patterns/patterns.html`

The patterns home page is a source for information about all aspects of patterns and pattern languages.

Portland Pattern Repository

URL: `http://c2.com/ppr/`

"The Portland Pattern Repository collects solutions to recurring problems found computer programming. Each solution is written as a stylized essay, most in the Portland Form. Some are linked into languages. All address the design of computer programs."

UIUC Smalltalk/Patterns Group

URL: `http://st-www.cs.uiuc.edu/users/smarch/index.html`

Projects include patterns, the refactoring browser, and a Smalltalk lint tool.

1.24 • What OOP publications have a net presence?

SIGS Publications

URL: `http://www.sigs.com/publications/sigspubs.html`

JOOP, *C++ Repor*t, *Object Magazine*, *Report on Object Analysis & Design*, *The Smalltalk Report* and more...

1.25 • What OOP publications are based on the net?

Object Currents

URL: `http://www.sigs.com/objectcurrents/`

"SIGS Publications presents Object Currents, the first significant hypermedia journal to provide developers and managers of object-oriented systems with a full range of articles covering the entire spectrum of object technology. Object Currents articles are accompanied by several articles from SIGS' journals, but are set apart by their ’eye on the future’ perspective. First priority is given to new and advanced topics of real importance to the object community, including applied research and development and new techniques and systems."

1.26 • Are there Smalltalk tutorials on the net?

GNU Smalltalk Tutorial

URL: `ftp://prep.ai.mit.edu/pub/gnu/smalltalk-tutorial.ps.gz`

URL: `ftp://prep.ai.mit.edu/pub/gnu/smalltalk-tutorial.txt.gz`

A tutorial on the Smalltalk language provided with the GNU Smalltalk implementation.

Smalltalk-in-the-Large Tutorial

URL: `http://www.bytesmiths.com/pubs/stitl/`

A tutorial on multi-person development with Smalltalk provided by ByteSmiths.

Smalltalk Textbook

URL: `http://sracog.srainc.com/~aoki/SmalltalkTextbook/index.html`

Written by Atsushi Aoki; translated by Kaoru Rin Hayashi and Brent N. Reeves.

Tutorial on ObjectWorks R4.1 (in French)

URL: `http://zeus.enst-bretagne.fr/Tutoriaux/smalltalk/s80part1/s80part1.html`

Smalltalk 4.1, un tutorial, A. Beugnard

"Nous présentons dans ce document l'interface de Smalltalk-4.1, les principes de programmations, les outils de développement et les classes de bases. Notre démarche est incrémentale et sollicite le lecteur. Nous vous conseillons de lire ce document activement, en face d'un environnement Smalltalk. De nombreux exercices sont proposés qui permettent de mettre en pratique les concepts et les outils décrits."

"Un Index en fin de document permet de retrouver rapidement la plupart des concepts, classes, méthodes décrits."

Tutorial on Model-View-Controller (MVC)

URL: `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`

This paper, by S. Burbeck, originally described the MVC framework as it existed in Smalltalk-80 v2.0. It was updated in 1992 to take into account the changes made for Smalltalk-80 v2.5. ParcPlace made extensive changes to the mechanisms for versions 4.x that are not reflected.

Frequently Asked Question

1.27 • What are the home pages of vendors and suppliers?

See "Smalltalk Implementations" on page 21 for net addresses of vendors and suppliers.

Frequently Asked Question

1.28 • What other Smalltalk sites are there on the Internet?

The Smalltalk Hints Index

URL: `http://www.amsinc.com/special/techexp/objtech/hints/index.htm`

Hints on Smalltalk programming as noted by the programming staff of AMS.

Smalltalk Industry Council (STIC)

URL: `http://www.webpress.net/stic/`

“STIC is a non-profit trade association whose goal is to promote the awareness of and increase demand for Smalltalk by: establishing Smalltalk as the object-oriented language of choice for application development in the enterprise; creating a focal point for the Smalltalk community; listening and responding to the needs of Smalltalk users; encouraging the participation of all segments of the Smalltalk industry (vendors, service providers and users); and encouraging standards for Smalltalk.”

Frequently Asked Question

1.29 • Are there Smalltalk User’s Groups?

Puget Sound Smalltalk Users’ Group maintains a list of other Smalltalk users’ groups.

The Puget Sound Smalltalk Users Group:

URL: `http://www.halcyon.com/podenski/stug/st-user-groups.html`

Frequently Asked Question

1.30 • What Smalltalk indexes are there?

Various people maintain lists of relevant Smalltalk and OOP information on the net. Here is a collection of such pages.

Aoki's Home Page

URL: `http://sracog.srainc.com/~aoki/`

English and Japanese Smalltalk goodies.

Bloomsbury

URL: `http://w3.win-uk.net/software/stlinks.htm`

A list of Smalltalk resources on the web.

Cocking and Co., UK

URL: `http://www.cocking.co.uk/`

Tips and technical articles, links to websites covering Smalltalk and other object-oriented topics, the Smalltalk User Group UK, and other Smalltalk-related information is provided by Cocking and Company.

DI-UFPE Smalltalk Page

URL: `http://www.di.ufpe.br/smalltalk/`

Pointers to Smalltalk information.

European Smalltalk User Group

URL: `http://www-laforia.ibp.fr/~fdp/esug.html`

"ESUG is a non-profit association that gathers European users of (all dialects of) Smalltalk. Its goals are to promote Smalltalk use; promote communication between Smalltalk users through meetings, periodic newsletter, software exchange, electronic mail; help and link local Smalltalk users groups; identify Smalltalk users and their needs; and represent them towards discussions with vendors."

Francois PACHET's Bookmarks, France

URL: `http://www-laforia.ibp.fr/~fdp/esug-bookmarks.html`

World wide coverage but with a distinctive European flavor.

Jeff McAffer's Smalltalk Page, Japan

URL: `http://web.yl.is.s.u-tokyo.ac.jp/members/jeff/smalltalk.html`

Jeff Sutherland's Object Technology Home Page, USA

URL: `http://www.tiac.net/users/jsuth/`

The Object Oriented Page

URL: `http://www.well.com/user/ritchie/oo.html`

A long list with Smalltalk and other pointers.

Smalltalk Developer's Site, USA

URL: `http://www.rwi.com/smalltalk/smalltalk.html`

Information provided by RothWell International.

The Smalltalk Store, USA

URL: `http://www.smalltalk.com`

The Smalltalk Store home page lists many Smalltalk resources.

Smalltalk with Tim and Linda, The Netherlands

URL: `http://www.htsa.hva.nl/~linda/index.html`

Grafische wandeling door de wereld van Smalltalk.

Software Composition Group Bibliography Server, Switzerland

URL: `http://iamwww.unibe.ch/cgi-bin/oobib?Smalltalk`

The Software Composition Group at the University of Bern, Institute of Computer Science and Applied Mathematics, maintains a server that searches for papers on object-oriented topics. This URL returns all such references containing the keyword ’`smalltalk`’.

Serge's Smalltalk Page, France

URL: `http://www.info.unicaen.fr/~serge/smalltalk80.html`

UIUC Sources of Smalltalk Information, USA

URL: `http://st-www.cs.uiuc.edu/other_st.html`

Yahoo’s Smalltalk Page

URL: `http://www.yahoo.com/Computers_and_Internet/`
`Programming_Languages/Smalltalk/`

Part II: Smalltalk Implementations

Each of the versions of Smalltalk is briefly described along with information about the source of the version and extensions provided by various vendors. Please note that the author works for one of these vendors†.

Descriptions of commercial Smalltalk vendors are limited to a brief summary of which versions and what platforms are supported (taken from the vendors WWW sites), and information on obtaining more details.

Descriptions of non-commercial versions of Smalltalk tend to be longer; again, the information is taken from the associated web sites.

The Smalltalk versions listed in this section are:

(Earlier versions of this FAQ attempted to list third-party vendors of Smalltalk products. Due to the increasing number of such vendors and the ever-changing product lists, this version of the FAQ has dropped the list. Thanks to all those who sent information.)

†. The author works for IBM but is not employed by the group which produces the IBM Smalltalk products.

ANSI Smalltalk

Frequently Asked Question

2.1 • What is ANSI Smalltalk?

ANSI Smalltalk is an incomplete and as-yet-to-be publicly proposed standard for Smalltalk. For information, see:

URL: `http://www.x3.org/tc_home/x3j20.html`

Frequently Asked Question

2.2 • What does ANSI Smalltalk cover?

The standard is expected to cover the language and base (non-graphics) classes including magnitudes and collections.

Frequently Asked Question

2.3 • What doesn't ANSI Smalltalk cover?

The standard is not expected to cover graphics, windows, or platform dependent functions.

Frequently Asked Question

2.4 • How do I get a copy of the ANSI Smalltalk standard?

There is no standard or publicly available draft at this time. A draft was available for review in early 1996 and additional drafts may be released later.

EMail: `x3j20@qks.com`

The X3J20 chairman is:

Contact:	Mr. Yen-Ping Shan
Address:	IBM Corporation PO Box 12195 Bldg. 062 RTP, NC 27709
EMail:	`shan@vnet.ibm.com`

Digitalk Visual Smalltalk

This section describes the Digitalk products sold prior to the merger with ParcPlace. Separate sections describe the ParcPlace products, and the new products of the merged company.

Frequently Asked Question

2.5 • What is Digitalk Visual Smalltalk?

Digitalk Visual Smalltalk is a series of related Smalltalk systems produced first by Digitalk, Inc., but now by ParcPlace-Digitalk.

Frequently Asked Question

2.6 • Who produces Digitalk Smalltalk?

Digitalk Smalltalk products are produced by ParcPlace-Digitalk.

URL:	`http://www.parcplace.com/`
Address:	ParcPlace-Digitalk, 999 E. Arques Avenue, Sunnyvale, CA 94086-4593
Phone:	Main: 408-481-9090 Sales: 800-759-7272 Tech Support: 800-253-3415, 408-773-7474
FAX:	Sales: 408-481-9095

Frequently Asked Question

2.7 • What versions are there of Digitalk Smalltalk?

URL: `http://www.parcplace.com/digitalk/home/vse.html`

Visual Smalltalk versions are available for Windows 3.1, Windows NT and OS/2. Each contains a visual application building tool. An extended product, Visual Smalltalk Enterprise, adds tools for team development.

Dolphin Smalltalk

Frequently Asked Question

2.8 • What is Dolphin Smalltalk?

Dolphin Smalltalk is a development system for PC based Microsoft Windows™ applications. Dolphin is intended to be a complete but low cost Smalltalk environment. The release price is not yet finalized but is likely to be under $200. During the Beta programme, Dolphin Smalltalk is available free from the vendors web site.

—Adapted from the web site

Frequently Asked Question

2.9 • Who produces Dolphin Smalltalk?

Dolphin Smalltalk is produced by Intuitive Systems Limited.

Address:	Broadlands House, Primett Road, Stevenage, Hertfordshire, SG1 3EE, United Kingdom.
Phone:	+44 (0) 1438 317966
FAX:	+44 (0) 1438 314368
EMail:	
URL:	`http://www.intuitive.co.uk/home.htm`

Frequently Asked Question

2.10 • What versions are there of Dolphin Smalltalk?

Dolphin Smalltalk runs on Windows.

Enfin Smalltalk

"VMARK Enfin Smalltalk" on page 35.

GemStone Smalltalk

GemStone is an object-oriented relational database system. In addition to supporting the commercial Smalltalk systems, it contains a Smalltalk language component as well.

Frequently Asked Question

2.11 • What is GemStone Smalltalk?

(more needed)

Frequently Asked Question

2.12 • Who produces GemStone Smalltalk?

GemStone Smalltalk is produced by GemStone Systems Inc.

Address:	15400 NW Greenbrier Parkway, Suite 280, Beaverton, OR 97006
Phone:	(503) 629-8383 (800) 243-9396
FAX:	(503) 629-8556
EMail:	`info@gemstone.com`
URL:	`http://www.slc.com/`

GNU Smalltalk

Frequently Asked Question

2.13 • What is GNU Smalltalk?

GNU Smalltalk is an implementation of Smalltalk provided by the Free Software Foundation, Inc. and subject to its ’copyleft’ license. It is free for individual use. It was written by Steve Byrne.

“GNU Smalltalk attempts to be a reasonably faithful implementation of Smalltalk-80 as described in the ’Blue Book’... . The syntax that the language accepts and the byte codes that the virtual machine interprets are exactly as they appear in the Blue Book.”

Frequently Asked Question

2.14 • How do I get GNU Smalltalk?

GNU Smalltalk can be downloaded from:

URL: `ftp://prep.ai.mit.edu/pub/gnu/`

Address: Free Software Foundation, Inc., 675 Mass Ave., Cambridge, MA 02139, USA

Frequently Asked Question

2.15 • What versions are there of GNU Smalltalk?

GNU Smalltalk runs on Apollo 3000/4000/10000, Atari, DECStation 3100, Encore Multimax, HP 9000, Interactive 386, Sony News 1810, Sun3, Sun4, NeXT, Pyramid, Sequent, SGI Iris-4D, Tektronix 431, and VAX. It is distributed in source form only.

IBM Smalltalk and VisualAge

Frequently Asked Question

2.16 • What are IBM Smalltalk and VisualAge?

IBM Smalltalk[†] and IBM VisualAge are commercial Smalltalk systems from IBM. VisualAge contains a visual application builder in addition to the Smalltalk development system provided in IBM Smalltalk. Each comes in versions with and without team development tools.

IBM VisualAge Home Page

URL: `http://www.torolab.ibm.com/software/ad/adwrad.html`

A web-based FAQ for IBM Smalltalk is at:

URL: `http://www.torolab.ibm.com/software/ad/vishints.html`

IBM VisualAge Fixes FTP Site

URL: `ftp://ps.boulder.ibm.com/ps/products/visualage/fixes/`

Where to get fixes for VisualAge, plus demos and other information.

†. Note that the author of this FAQ works for IBM.

Frequently Asked Question

2.17 • Who produces IBM Smalltalk?

IBM. Contact your local IBM office, or:

URL:	`http://www.ibm.com/`
EMail:	`askibm@info.ibm.com` (Within the USA only.)
Phone:	1-800-IBM-3333 International: +1 520 574 4600 (Ask to be connected to the IBM Information Service Center.)

Frequently Asked Question

2.18 • What versions are there of IBM Smalltalk?

IBM Smalltalk comes in versions for OS/2, Windows, and AIX.

Little Smalltalk

Frequently Asked Question

2.19 • What is Little Smalltalk?

Little Smalltalk is *not* a Smalltalk-80 system, but is an experiment in building a minimal Smalltalk system. The source code for the interpreter is less than 1800 lines of code. The image consists of less than 4000 objects. It runs in almost no memory, it's reasonably fast, it's easy to understand, easy to modify, and runs on Unix. In order to get such a small Smalltalk to run there were some changes:

- No graphics.
- The user interface is a real-eval-print loop rather than a browser, but it is written in Smalltalk, so it can be changed
- Primitives and cascades are not compatible.
- The standard class hierarchy differs from Smalltalk-80; the basic collections are List, Tree, Dictionary, Array and String.
- The only numbers are large and small integers.

—From the web site

In addition, there are Mac and Windows versions available from other parties.

Frequently Asked Question

2.20 • Where can I get Little Smalltalk?

The web page for Little Smalltalk is:

URL: `ftp://ftp.cs.orst.edu/users/b/budd/little/index.html`

The current (version 3) release for UNIX:

URL: `ftp://ftp.cs.orst.edu/pub/budd/small.v3.tar`

The current (version 3) release for the Macintosh:

URL: `ftp://sumex-aim.stanford.edu/info-mac/dev/little-smalltalk-315.hqx`

The version 4 beta for UNIX:

URL: `ftp://ftp.cs.orst.edu/users/b/budd/little`

Windows Little Smalltalk Interface is MS Windows 3.1 application built over the original Little Smalltalk interpreter. Preserving as much as possible from the original interpreter it provides a graphic user interface which makes it much easier to develop programs. The package was developed in the Advanced Programming Lab 2, Department of Computer Science, Technion -- Israel Institute of Technology.

URL: `ftp://ssl.cs.technion.ac.il/pub/Projects/`

The code is mirrored at:

URL: `ftp://ftp.smalltalk.com/pub/little.smalltalk/`

Object Connect's MT Smalltalk

Frequently Asked Question

2.21 • What is Smalltalk MT?

"Smalltalk MT is a high-performance implementation of the popular Smalltalk language. Smalltalk MT targets Windows NT 4.0 and features an optimizing compiler that generates fast, compact code."

"Smalltalk MT combines the productivity of a true object oriented language and an interactive development environment with the performance edge and functionality of fully compiled code. Programs built with Smalltalk MT are very fast and small, allowing developers to distribute them over the net. Both executables (EXE files) and dynamic link libraries (DLLs) can be built, and the result is a genuine PE (portable executable) file. With its extensive Win32 support, it is easy to create state-of-the-art Windows applications. Smalltalk MT supports advanced features such as true multithreading and native exception handling."

Frequently Asked Question

2.22 • Who produces Smalltalk MT?

Smalltalk MT is produced by Object Connect.

Address:	Object Connect, SARL 42, rue de Tauzia, 33800 Bordeaux, France
URL:	`http://www.objectconnect.com/`

Object Technology International

Frequently Asked Question

2.23 • What is OTI?

Object Technology International Inc. (OTI) is a software engineering company specializing in Smalltalk and other object-oriented technologies. As of early 1996 it is a wholly owned subsidiary of IBM. Products include the ENVY product, Smalltalk systems for imbedded applications, and IBM Smalltalk.

Frequently Asked Question

2.24 • Where is OTI?

URL:	`http://www.oti.com/`
Address:	Object Technology International Inc., 2670 Queensview Drive, Ottawa, Ontario, Canada K2B 8K1
Phone:	(613) 820-1200
FAX:	(613) 820-1202
EMail:	`info@oti.com`

ParcPlace VisualWorks Smalltalk

This section describes the ParcPlace products sold prior to the merger with Digitalk. Separate sections describe the Digitalk products, and the new products of the merged company.

Frequently Asked Question

2.25 • What is VisualWorks Smalltalk?

URL: `http://www.parcplace.com/marketing/products/vwfamily.html`

ParcPlace VisualWorks Smalltalk is a series of related Smalltalk systems produced first by ParcPlace, Inc., but now by ParcPlace-Digitalk.

Frequently Asked Question

2.26 • Who produces VisualWorks Smalltalk?

ParcPlace VisualWorks Smalltalk products are sold by ParcPlace-Digitalk.

URL:	`http://www.parcplace.com/`
Address:	999 E. Arques Avenue, Sunnyvale, CA 94086-4593
Phone:	Main: 408-481-9090 Sales: 800-759-7272 Tech Support: 800-253-3415, 408-773-7474
FAX:	Sales: 408-481-9095

Frequently Asked Question

2.27 • What versions are there of VisualWorks Smalltalk?

VisualWorks runs on the following platforms: MS-Windows, Windows NT, OS/2, Macintosh, SGI, SunOS 4, Solaris 2, IBM RS/6000, HP, and Sequent.

ParcPlace-Digitalk Smalltalk

This section describes the Smalltalk products resulting from the merger between Digitalk and ParcPlace. Separate sections describe the products of ParcPlace and Digitalk prior to the merger.

Frequently Asked Question

2.28 • What is ParcPlace-Digitalk Smalltalk?

The Smalltalk systems described in this section are the new ’merged’ products incorporating technology of both Digitalk and ParcPlace Systems. These products have not been released.

Frequently Asked Question

2.29 • Who produces ParcPlace-Digitalk Smalltalk?

ParcPlace-Digitalk Smalltalk products are produced by ParcPlace-Digitalk.

URL:	`http://www.parcplace.com/`
Address:	999 E. Arques Avenue, Sunnyvale, CA 94086-4593
Phone:	Main: 408-481-9090 Sales: 800-759-7272 Tech Support: 800-253-3415, 408-773-7474
FAX:	Sales: 408-481-9095

Frequently Asked Question

2.30 • What versions are there of ParcPlace-Digitalk Smalltalk?

None at this time.

QKS SmalltalkAgents

Frequently Asked Question

2.31 • What is QKS Smalltalk/Agents?

Quasar Knowledge Systems, Inc. (QKS) SmalltalkAgents is a Smalltalk-dialect and development system. While very similar to Smalltalk, the language and class libraries differ in many details.

Frequently Asked Question

2.32 • Who produces QKS Smalltalk/Agents?

Quasar Knowledge Systems, Inc.

URL:	`http://www.qks.com/`
Address:	Quasar Knowledge Systems, Inc., 9818 Parkwood Dr., Bethesda, MD 20814
Phone:	Sales: 1-800-296-1339 Support: 301-530-4853 International: +1 301 530 4853
FAX:	1-301-530-5712
EMail:	Sales and ordering info: `sales@qks.com` International contact info: `international@qks.com` QKS technical support: `support@qks.com`

Frequently Asked Question

2.33 • What versions are there of QKS Smalltalk/Agents?

“Macintosh 68k edition is available now. Macintosh Universal edition (for 68k and PPC development) will be available in Q2 of 1996. Windows edition will be available in Q4 1996."

SELF Smalltalk

Frequently Asked Question

2.34 • What is SELF/Smalltalk?

“The Self 4.0 Smalltalk system comprises a translator, written in Self, which translates Smalltalk code to Self code, and a set of Smalltalk classes. Most of the Smalltalk classes are based on those included in GNU Smalltalk 1.1.1. There are ’core’ classes for collections and magnitudes, but little else; this is not intended to be an industrial-strength Smalltalk system. The Self 4.0 Smalltalk System has been built for several purposes:

- to provide a freely-available system that may be used to teach Smalltalk,
- to hearten Smalltalk programmers to the prospect of learning Self by demonstrating the linguistic proximity of Self to Smalltalk, and
- to showcase Self 4.0's adaptive optimization technology.”

—From the web site

Frequently Asked Question

2.35 • Where can I get SELF/Smalltalk?

SELF-Smalltalk

URL: `http://self.smli.com/release/smalltalk.html`

Self Itself

URL: `http://self.smli.com/`

Frequently Asked Question

2.36 • What versions of SELF/Smalltalk are there?

Self currently runs on SPARC-based Sun workstations running SunOS 4.1.x, Solaris 2.3, or Solaris 2.4.

Smalltalk-X

Frequently Asked Question

2.37 • What is Smalltalk-X?

“Smalltalk/X is a new and complete implementation of the Smalltalk language and consists of both an integrated environment for program development and a stand-alone smalltalk compiler, generating true machine code. Its features are:

- Language syntax and semantic compatible to the industry standard;
- a comprehensive library of basic classes, including lightweight processes and exception handling;
- many user interface widget classes offering parametrized view styles; and
- an integrated programming environment including browsers, monitors and symbolic debugger for efficient program development.”

“ST/X is a product of Claus Gittinger, Development & Consulting. Packaging and distribution is performed by independent distributors. A free (somewhat stripped down) version for educational use is also available via FTP from various sites.”

—From the web site

Frequently Asked Question

2.38 • Where can I get Smalltalk-X?

URL: `http://www.informatik.uni-stuttgart.de/stx/stx.html`

Frequently Asked Question

2.39 • What versions are there of Smalltalk/X?

The free versions include: Solaris 2.4, Linux 1.1.59, Irix 5.2, Ultrix (MIPS) 4.4, and NextStep 2.1.

VMARK Enfin Smalltalk

Frequently Asked Question

2.40 • What is Enfin Smalltalk?

"VMARK's ObjectStudio provides a complete, object-oriented, rapid application development environment for designing, developing, and running Windows-, OS/2-, and UNIX-based client/server applications. ENFIN Smalltalk is a comprehensive object-oriented development environment that forms the foundation of ObjectStudio. It includes the Smalltalk language, Class Browser, complete debugging facilities, GUI Designer, connectivity tools, and a variety of business tools."

Frequently Asked Question

2.41 • Who produces Enfin Smalltalk?

URL:	`http://www.vmark.com/`
Address:	VMARK Software, Inc. 50 Washington Street, Westboro, MA 01581-1021
Phone:	(508) 366-3888 Technical Support: 1-800-729-3553
FAX:	(508) 366-3669 Technical Support: (508) 389-8750

Frequently Asked Question

2.42 • What versions are there of Enfin Smalltalk?

Enfin Smalltalk supports Windows, OS/2, AIX, Solaris, and HP-UX.

Part III: Definitions of Terms

Many terms in Smalltalk are also used in other languages such as C++, Eiffel, Java and Simula. While there may be a surface similarity, the terms often mean quite different things. This chapter defines many common terms in the context of Smalltalk.

Definitions of Terms: Objects

Frequently Asked Question

3.1 • What is an object?

In Smalltalk, an *object* is a collection of data and code. The data is hidden within the object and cannot be accessed except through code in the object.

Frequently Asked Question

3.2 • What is a method?

A method is a Smalltalk subroutine, no more and no less. It always takes at least one parameter (`self`) and always returns one value. Methods always belong to a class.

Frequently Asked Question

3.3 • What is a class?

A *class* is the definition of an object. The class of an object holds a list of the data that an object will have, and it holds the code for the object. Each different kind of object has its own class.

Smalltalk programs are created by writing classes. Each class defines data and a set of methods, the code, which operates on that data. Figure 1 illustrates a class and two instances.

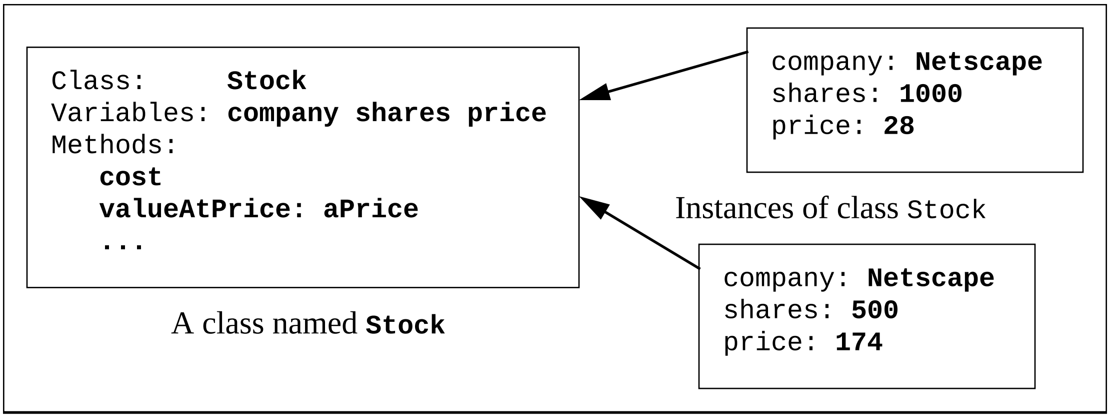

Figure 1: A Class and Instances. A class named `Stock` is illustrated on the left, and two instances of the class are shown on the right. The class contains a list of the variables, `company`, `shares`, and `price`, that each instance of the class will have. The class also contains code that each of the instances will share, in this case including `cost` and `valueAtPrice:`.

Frequently Asked Question

3.4 • What is an instance?

An *instance* of a class is simply an object. It holds data and a pointer to the class which holds the code.

Frequently Asked Question

3.5 • What is a subclass?

A *subclass* is a class that is defined using another class as a basis. The other class is called the *superclass* (or parent class). The subclass adds to the implementation (or shares the implementation) of the superclass.

Subclassing in Smalltalk is implementation sharing not subtyping.

See [4.41] *'What is a type?'* on page 68, [4.42] *'Does Smalltalk have types?'* on page 69, and [4.43] *'Is there any disagreement about types in Smalltalk?'* on page 71.

Frequently Asked Question

3.6 • What are encapsulation and information hiding?

Encapsulation and information hiding refer to the act of hiding data behind a wall of code. Encapsulation is a basic characteristic of all OOP languages and of some non-OOP languages.

Frequently Asked Question

3.7 • What is a hierarchy?

A hierarchy is an outline or tree of class definitions. In Smalltalk, the top (or root) of the hierarchy is a very general class named `Object`. All other classes are subclasses of `Object` or of some other class (which must be a subclass of Object or another class).

There are some weird exceptions; see xxxxxxx.

Frequently Asked Question

3.8 • What is polymorphism?

Polymorphism means multiple meanings. In Smalltalk, polymorphism means that there are multiple methods with the same name and that any one of them might be invoked by a single message send.

Example

A group of classes define various kinds of employees. There is one class for retired employees, one for active employees, one for executives, one for part time employees, and so on.

Each class might have a `computePaycheck` method to compute the amount of an employee’s paycheck, but the methods would be different because of the differing requirements of each kind of employee.

At the point where a paycheck amount needs to be computed, the `computePaycheck` message is sent to an instance of one of the employee classes:

```
    anEmployee computePaycheck
```

The method actually invoked depends entirely on the object held in `amEmployee`.

Definitions of Terms: Messages

Frequently Asked Question

3.9 • What is a message?

A message is an operation that can be requested of an object. It consists of the name of the message, which is the name of a method, and values for any associated parameters. All messages are considered to have at least one parameter, the value that will become `self`.

Now, consider:

```
floob zotFarb: aValue
```

If the object held in `floob` responds to a method named `zotFarb:` then these are the parts of the message:

1. The name `zotFarb:`
2. The value that will be `self` (the value held in `floob`)
3. A value to pass for its one parameter.

Frequently Asked Question

3.10 • What is a message send?

A message send is the process of invoking a method in an object using a message. It has two steps: finding the method, and invoking the method.

Finding the method

The name of the message is compared with methods in the target object. If not found, then each of the parents are examined successively until one is found that has such a method. (If no method is found, an error condition is raised.) In general, the object to which the message is sent is not known until runtime and this lookup process must logically occur each time a message is sent.

Invoking the method

Once the appropriate method is found, then the method is invoked just as if it were a procedure in a procedural language. Parameters are bound to their local names, the stack is appropriately massaged, and execution begins.

Upon termination, control returns to the method that sent the message at the point just after the message send. A value is always returned although the sender has the option of ignoring it.

Frequently Asked Question

3.11 • How does a message send differ from a procedure call?

A message send is identical to a procedure call once a method is found to invoke.
See See *3.10 'What is a message send?'*

Definitions of Terms: Classes

Frequently Asked Question

3.12 • Are classes objects?

Yes; classes in Smalltalk are objects and are instances of some other object.
See "Metaclasses" in section "Implementation" on Page 170.

Frequently Asked Question

3.13 • What is an abstract class?

An abstract class is just a class; Smalltalk makes no distinction between abstract classes and any other kind. However, programmers find it useful to define classes that are not complete, and that will never have instances. Such classes define some general concept and some protocol (a set of methods) that all its subclasses will inherit.

The base class library contains a number of abstract classes, or abstract superclasses, as they are sometime called. These include:

Class	**Abstracts**
`Object`	The basic idea of an object: existence, size, ...
`Magnitude`	Values that compare uniquely with < and >
`Number`	Values that can be added, subtracted, ...
`Stream`	Sequences of values
`Collection`	Groups of values

Part IV: The Smalltalk Language

Literals

Literals are objects written literally, that is, with actual characters on the page or screen. Since the value of a literal is always the same, literals are sometimes called constants or literal constants.

Smalltalk has literal constants for integers, floating point numbers, a limited set of fractions, characters, booleans, strings, symbols, and arrays with literal contents.

Frequently Asked Question

4.1 • What are the forms of integer literals?

Integers have a rich set of literal constant forms which allow the description of values with bases ranging from 2 to 36 and in arbitrary lengths.

The simplest integer literal consists of one or more decimal digits with an optional leading minus sign.

```
1   123   -3
```

There is no practical limit to the length of an integer constant.†

```
1248764358763487634987634987643598764598736598761348762348761 32
```

A number base can be specified by a prefix with a decimal number in the range 2 to 36 followed by a lower case ’`r`’, and followed by one or more of the digits allowed for that base. The values in each line below are equal to others on the same line:

```
2r11111111   4r3333   8r377   10r256   16rFF   32r7V
36rSMALLTALK   80738163270632
```

The characters allowed for base *n* are the *n* leading characters of:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

In addition, VisualWorks allows lowercase letters.

Exponents

Except in VisualWorks, integers may also have an exponent, a lowercase ’`e`’ followed by one or more decimal digits. (In VisualWorks, the exponent indicates a floating point literal.) The values in each line are equal to others on the same line:

```
1     1e0
300   2r11e2   3e2   4r3e2
```

See *4.3 ’What are the forms of fraction literals?’* for information on negative exponents.

Note the following cases which do not always produce quite the results that might be expected:

`16r2E1`	Produces the value `737`; the ’`E`’ is uppercase and isn’t an exponent.
`16RFF`	Tries to send the `RFF` message to `16` since the ’`R`’ is uppercase, or get a message about digit too big, depending on the vendor.
`16r2abff`	Tries to send the `abff` message to `2`; ’`abff`’ is lowercase. (In VisualWorks it answers 175103.)

Frequently Asked Question

4.2 • What are the forms of floating point literals?

Floating point numbers have a rich set of literal constant forms which allow, in most implementations, the description of values with bases ranging from 2 to 36 and, in some implementations, values in several precisions.

†. The actual limits depend on the implementation. A typical implementation might keep a 16-bit or a 32-bit count of the number of 32-bit words which make up the long integer. Execution time and memory size are the real limiting factors.

The simplest floating literal consists of one or more decimal digits with an optional leading minus sign and an *imbedded* decimal point.

```
1.0   12.3   -3.0
```

A leading or trailing decimal point is taken as a statement separator. These are *not* floating point constants:

```
123.   .123
```

VisualWorks does not require the decimal point if an exponent is present:

```
123e2   12300.0
```

There is no practical limit to the length of an floating point constant†, but only as much of the constant as can be represented in the implementation will be retained at run time. It is sometimes useful to code constants at a greater precision to allow for portability to platforms with a greater precision.

```
3.14159265358979323846264338327950288419716939937510582097494459
```

Number Bases

A number base can be specified by a prefix with a decimal number in the range 2 to 36 followed by a lower case `’r’`, and followed by one or more of the digits allowed for that base. (But note that VisualWorks does not allow non-decimal floating-point numbers.) The values in each line are equal to others on the same line:

```
2r11111111.0  4r3333.0  8r377.0  10r256.0  16rFF.0   32r7V.0
36rSMALL.TALK   48069417.81373362
```

The characters allowed for base *n* are the *n* leading characters of:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Exponents

Floating point values may also have an exponent, which is a lowercase `’e’` (and `’d’` in VisualWorks) followed by one or more decimal digits. (VisualWorks also allows upper case `’D’` and `’E’`.) The values in each line are equal:

†. But, since only the first 70 or so digits are meaningful even if a quad-length value is supported, an implementation might not have tested anything much longer.

```
   1.0     1.0e0
   300.0   2r11.0e2   3.0e2   4r3.0e2
```

In VisualWorks, the `’e’` or `’E’` exponent indicates a short precision floating point number and the `’d’or ’D’` indicates a double-precision floating-point number. (Unconfirmed reports indicate that a quad-precision constant may be available in VisualWorks 2.5.)

In other implementations, the precision is platform and implementation dependent. See “Floating Point” on page 137 for more information on floating point numbers.

Frequently Asked Question

4.3 • What are the forms of fraction literals?

IBM Smalltalk

The only fraction literals are almost accidental and come from allowing signed exponents on integers.

`123e-4`	Equivalent to the expression: `(123/1000)`

VisualWorks

There are no fraction literals.

`123e-4`	Produces the result: `0.0123`

Frequently Asked Question

4.4 • What is the form of a Decimal literal?

Decimal support varies from implementation to implementation.

IBM Smalltalk

There are no decimal literals. Decimal numbers are written as string literals and converted to decimal with the `asDecimal` message.

```
   ’12.345’ asDecimal
```

VMARK Enfin Smalltalk

Enfin Smalltalk provides a decimal literal consisting of a number with a decimal point, suffixed with the character `’d’`.

```
12.345d
```

Draft ANSI Standard

The draft ANSI Standard proposes a scaled-decimal literal consisting of a number with a decimal point, suffixed with the character ’s’.

`12.345s`	A default number of digits
`12.345s10`	Specify 10 digits

Frequently Asked Question

4.5 • What is the form of a character literal?

Character literals consist of a dollar sign followed by a character. A dollar sign followed by a blank is the blank character, and followed by a dollar sign is a dollar sign.

`$a   $A   $z   $9   $.   $(   $)   $` (a blank)   `$$`

The character following the dollar sign can be any character in the implementation character set, including linefeed or return, but only the ASCII characters in the range 32 to 127 are guaranteed to be portable between implementations and platforms.

Frequently Asked Question

4.6 • What is the form of a string literal?

A string literal is a pair of single quote marks, possibly enclosing other characters.

```
   ’’   ’a’   ’A String of Characters’
```

A single quote mark is represented by two successive single quote marks.

```
   ’It isn’’t too hard to do.’
```

Note that a string with one character is not a character. These are not equal:

```
   ’a’   $a
```

Example 1, Display

Frequently Asked Question

4.7 • What are the boolean literals?

The boolean literals are `true` and `false`.

The value `true` is not the same as the class `True`; it is an instance of class `True`. The value `false` is not the same as the class `False`; it is an instance of class `False`. Thus, this code will fail with a message about the receiver not being a Boolean:

```
| bool |
bool := False.
bool ifFalse: [ Transcript cr; show: 'False']
```

Example 2, Display

The correct code is:

```
| bool |
bool := false.
bool ifFalse: [ Transcript cr; show: 'false']
```

Example 3, Display

While `true` is an instance of `True`, it is a special one and other instances of `True` cannot be substituted. The same also holds true for `false`.

```
" A new instance of True "
True basicNew ifTrue: [ 1 ]  ifFalse: [0]
" On some implementations, gets a message
  about not being a Boolean "
```

Example 4, Display

```
" A new instance of True "
True basicNew = true
" Answers false "
```

Example 5, Display

Frequently Asked Question

4.8 • What are the forms of symbol literals?

Symbols are similar to strings but have additional properties.

See “Symbol Questions” on page 131.

Symbol literals are made up of a pound or number sign (’`#`’) followed by a message selector.

```
#==   #abc   #negated   #to:by:   #in:the:course:of:
```

Since valid binary selectors are made up of one or two special characters, the effect of writing three or more characters is implementation defined. For example, IBM Smalltalk takes #=== as the two successive symbols #== and #= while Digitalk Smalltalk/V-Mac considers #=== as error.

Extended Symbol Literals

IBM Smalltalk and Enfin Smalltalk allow an alternate form of symbol literal consisting of a pound (or number) sign followed by a string. The examples above would look like this:

```
#'=='   #'abc'  #'negated' #'to:by:'  #'in:the:course:of:'
```

However, any character that is valid in a string can be in the extended symbol literal.

```
#' '   #'-----'   #'#'   #'A big step for mankind.'
```

An extended literal of the form:

```
#'aSymbolString'
```

produces a symbol with the same value as the expression:

```
'aSymbolString' asSymbol
```

Frequently Asked Question

4.9 • What is the form of an array literal?

An array literal is formed from a pound (or number) sign and a pair of parentheses enclosing numeric literals, valid message selectors, character literals, string literals, and other array literals. (Not included are boolean literals and extended symbol literals.) Nested arrays need not have the leading pound sign.

`#( 2 3 4)`	An array of integers
`#('abc' 'def')`	An array of strings
`#(2 $c 7.3 'def' 2r1111)`	An array holding various objects
`#( #(1 2) #(3 4) #(5 6))`	An array of arrays
`#((1 2) (3 4) (5 6))`	The same array of arrays

IBM Smalltalk

IBM Smalltalk supports byte array literals which are formed from a pound (or number) sign and a pair of square brackets enclosing integer literals in the range 0-255.

`#[0 1 2 3 255 7]`	A byte array of 6 values

Enfin Smalltalk

VMARK's Enfin Smalltalk supports an alternative array literal using left and right curly brackets.

`{ 'hello world' $z 123.456 }`	
`{ #aSymbol #+ #at:put: }`	Pound-sign's required on symbols
`{ { 1 3 } { 4 5 } }`	Nesting of arrays

Variables and Names

Frequently Asked Question

4.10 • What characters can be used in names?

The following characters can be used in names.

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 _
```

Most implementations support an underscore character as if it were an alphabetic character. However, some support underscore as an alternate way of writing assignment; such usage is obsolete though.

See "Can I use an underscore character?" on page 56 for more information.

Names must start with an uppercase or lowercase letter (or an underscore where allowed).

Frequently Asked Question

4.11 • Are there conventions for naming classes?

Yes. Classes are virtually always named as one or more words in some natural language. The variable must start with an uppercase letter. By convention, each additional word starts with a capital letter and underscores are not used.

```
SortedCollection   LargeLandMammal   RetiredEmployee
```

Related sets of classes are often named with some common prefix or suffix.

```
Collection   SortedCollection   OrderedCollection
WriteFileStream   ReadFileStream   ReadWriteFileStream
```

Frequently Asked Question

4.12 • Are there conventions for naming methods?

Yes. Methods are virtually always named as one or more words in some natural language, or a standard term or abbreviation. By convention, each word but the first starts with a capital letter and underscores are not used.

```
sqrt   negated   min:   asFloat   employeeSpouseName   isEmpty
```

Methods with keyword selectors are named so that the part of the keyword selector that precedes a parameter names or implies that value.

```
on:from:to:   when:do:   translateBy:   truncateTo:   x:y:
indexOfSubcollection:startingAt:ifAbsent:
```

Frequently Asked Question

4.13 • What different kinds of variables are there?

There are ten kinds of variables in Smalltalk. Four kinds have some amount of global scope:

`Smalltalk` Global Variables

Global to all classes (in dictionary `Smalltalk`). See question *[4.23]*.

Class variables

Global to a class. See question *[4.26]*.

Pool variables

Global to classes that use the pool dictionary. See question *[4.30]*.

Special variables: `nil`, `true`, `false`, `self`, `super`, `Smalltalk`, and `thisContext`

Global to all classes; '`thisContext`' is available in VisualWorks only.

The other six kinds are local variables.

Instance variables

Specific to each instance

Class instance variables

Specific to class methods and subclasses.

See [4.27] *'What is a class instance variable?'* on page 59.

Method parameters

Local to the method

Method local variables

Local to the method

Block parameters

Local to the block, except that some older implementations make block parameters local to the method.

Block local variables

Local to the block, except that some older implementations make block parameters local to the method.

See [4.50] *'Do blocks differ between implementations?'* on page 76.

In a method, instance variable names, method parameters and locals must be unique, and must not be the same as block parameters and locals. None can be the same as a special variable. Different blocks in the same method may have the same parameter and local variable names as other blocks in the method.

```
method: parm1 with: parm2
   | temp1 temp2 |
   temp1 := instVar1.
   self flareWith: [ :bparm |
      | btemp |
      btemp := pbarm + instVar1 ].
   self flareWith: [ :bparm2 |
      | btemp2 |
      btemp2 := pbarm2 + instVar1 ].
```

The variables `parm1`, `parm2`, `temp1`, `temp2`, and `instvar1` must all be unique, and must be different from `bparm`, `bparm2`, `btemp`, and `btemp2`. However, it would be valid to have `bparm2` be the same as `bparm`, and `btemp` the same as `btemp2`.

4.14 • What is the lifetime of these variables?

In a development envrionment in which the image is repeatedly saved, the variables have the lifetime shown below. In a packaged application, variable lifetime ends when the application ends.

`Smalltalk` Globals

Forever; the variables can be deleted from the dictionary, thus shortening their ’lifetime’.

Class variables

The life of the class; the variables can be deleted from the class, thus shortening their ’lifetime’.

Pool variables

The life of the class or pool; the variables can be deleted from the pool, thus shortening their ’lifetime’.

Special variables

Forever.

Instance variables

The life of the instance.

Class instance variables

The life of the class; the variables can be deleted from the class, thus shortening their ’lifetime’.

Method parameters

The life of the method context.

Method local variables

The life of the method context.

Block parameters

The life of the block invocation, except for implementations not supporting block local variables, in which case it is the life of the method context.

Block local variables

The life of the block invocation, except for implementations not supporting block local variables, in which case it is the life of the method context.

Frequently Asked Question

4.15 • What is the search order for global variables?

Global variables are found by searching first for a class variable, then searching pool dictionary list in the order specified in the class definition, and then looking in the dictionary `Smalltalk`.

Frequently Asked Question

4.16 • Are there conventions for naming variables?

Instance variables are usually named either by the class of the data, or by the expected use or actual contents, or sometimes both. Skublics† calls these typed and semantic names.

Examples of typed names:

`aString`	An instance of `String`
`anInteger`	An instance of `Integer` or one of its subclasses
`aNumber`	An instance of `Number` or one of its subclasses

Examples of semantic names:

`numberOfBurners`	The number of burners (of a stove or hot air balloon)
`nameOfEmployee`	Some object holding an employee name

Examples of combined typed and semantic names:

`employeeNameString`	An instance of `String` holding an employee name
`burnerCountInteger`	An instance of `Integer` (or of a subclass)

Frequently Asked Question

4.17 • Are there conventions for naming instance variables?

Instance variables hold the data for classes and are defined in the class itself. Since there is little context from which to imply a meaning, the names should be

†. See *Smalltalk with Style* in question [1.7].

descriptive of the purpose and use of the variable. Thus, instance variables should be named using semantic names.

`employeeName`	An employe name
`numberOfCylinders`	A count
`lotSizeInAcres`	A number of some kind, giving the lot size in acres

Frequently Asked Question

4.18 • Are there conventions for naming class variables?

Class variables hold various bits of data, and are defined in the class itself. Since there is little context from which to imply a meaning, the names should be descriptive of the purpose and use of the variable. Thus, class variables should be named using semantic names.

`MaximumCylinderCount`	The most!
`NextEmployeeNumber`	What’s to come next

See [4.26] *’What is a class variable?’* for more information.

Frequently Asked Question

4.19 • Are there conventions for naming method parameters?

Method parameters can be named using either typed or semantic naming, depending on context. Typed names are useful when a value is of some expected class and context provides semantic clues:

```
cylinders: anInteger
   anInteger > MaximumCylinderCount ifTrue: [ "error" ].
   numberOfCylinders := anInteger
```

Semantic names are useful when context either does not provide semantic clues or the code is clearer anyway. Compare:

```
taxOnBuildingLot: lotSizeInAcres
   ^ lotSizeInAcres * self buildingLotTaxRate
```

with:

```
taxOnBuildingLot: aNumber
   ^ aNumber * self buildingLotTaxRate
```

Frequently Asked Question

4.20 • Are there conventions for naming temporary variables?

Local variables are best if named with a meaningful semantic name that indicates how the variable is used.

Some authors recommend using a new variable and a new name for each different use, rather than naming variables with simple names like `n` or `temp` and reusing them.

```
    (need example)
```

Example 6, Display

Frequently Asked Question

4.21 • When can variables have leading capital letters?

Leading upper-case letters are required on global variables (as defined in the dictionary `Smalltalk`), class variables, and pool variables. They are optional on other variables.

Some implementations make exceptions:

Implementation	**Must be lowercase**
Digitalk	Local variables, method & block parameters
IBM Smalltalk	(none)
ParcPlace VisualWorks	(none)
VMARK Enfin	(none, but see below)

VMARK Enfin Smalltalk does not require globals to start with an uppercase letter but does issue a warning if globals are defined with a lowercase letter.

Frequently Asked Question

4.22 • Can I use an underscore character?

Some implementations of Smalltalk allow an underscore character to be used in variable names. Some implementations still treat an underscore as an alternative for the assignment operation.

Implementation	**Is Underscore Allowed in Variables**
IBM Smalltalk	Yes; may be leading character; acts as lowercase
ParcPlace VisualWorks	Yes.
Digitalk	No.
VMARK Enfin	Yes.

Global Variables

Frequently Asked Question

4.23 • What is a global variable?

A global variable is a variable that is available to all classes. Global variables are created by making entries in the dictionary Smalltalk. In most implementations, classes are implemented as global variables; the class `Bottle` would be stored in the dictionary `Smalltalk` using the key `#Bottle`.

See [4.13] *'What different kinds of variables are there?'*.

Frequently Asked Question

4.24 • When should global variables be used?

Most experts say to never use a global variable! In Smalltalk, it is particularly bad to use variables defined in the dictionary `Smalltalk` because they are global to everything. Only classes and pool dictionaries should be global.

Note: Some software development tools provide for namespaces in Smalltalk which limit the visibility of global variables to sections of a class library.

Frequently Asked Question

4.25 • What alternatives are there to global variables?

Class variables

Instead of a global variable, make a class variable with the same name and set its value when the class is initialized.

Class instance variables

Instead of a global variable, make a class instance variable with the same name and set its value when the class is initialized. Implement a class method to answer its value.

See [4.28] *'What good is a class instance variable?'* on page 59.

Messages to classes

Instead of a global variable, use a class method. Have it return the same value as the global variable would have held. Values returned by methods are hidden. The method can be overridden by subclasses and a different value might be answered.

Some authorities recommend using class methods to answer constant values rather than using class variables:

```
maximumCylinderCount
    ^ 12
```

Separate classes

Some authorities recommend using a new class to hold values like the next employee number. The class, say `EmployeeNumber`, would have one instance per sequence of employee numbers. The next employee number would be thus be an instance variable.

(Need reference to Singletons in part XI)

Class Variables and Class Instance Variables

Frequently Asked Question

4.26 • What is a class variable?

A class variable is a special variable which is global to the instance and class methods of a class and all of its subclasses. Class variables must start with an uppercase letter and are a part of the definition of the class itself.

Class variables are used as alternatives to global variables when the needed scope is a class, its subclasses, and their instances.

Class variables are typically created by adding their name(s) to the class definition message displayed in a class browser. In Example 7 one class variable, `KnownPrimes`, is defined.

```
Object
  subclass: #SimplePrimeNumberGenerator
  instanceVariableNames: 'highest'
  classVariableNames: 'KnownPrimes'
  poolDictionaries: ''
```
Example 7

Frequently Asked Question

4.27 • What is a class instance variable?

A class instance variable is an instance variable of the class itself. It belongs to the class and its subclasses. It cannot be seen by instances of the class. It follows the normal rules for instance variables and can have a leading uppercase letter only if generally allowed in the implementation.

Class instance variables are typically created by adding their name(s) to the class definition message displayed in a class browser. In Example 8, one class instance variable, banana, is defined.

```
MonkeyBusiness class
  instanceVariableNames: 'banana'
```
Example 8

(Need example from IBM Smalltalk)◄

Frequently Asked Question

4.28 • What good is a class instance variable?

Class instance variables are rarely used by applications even though they have several advantages:

Information hiding

Instances of the class or of its subclasses cannot access the variable directly.

Overriding in subclasses

Subclasses can override the method, providing modified access or hiding access.

Replacing class variables

One possible use is to hold a value which is returned by a class method, rather than use a class variable to hold the same value. Instead of writing:

```
MySpecialValue
```

to access a class variable, you might write:

```
self class mySpecialValue
```

which invokes the class method `mySpecialValue` which answers the value of a class instance variable holding the same value. To set the value, you might write:

```
self class mySpecialValue: aValue
```

Storing information about instances

Another possible use for a class instance variable is to store information about instances. For example, a class that responds to `new:` might keep a table showing the distribution of sizes of instances.

Pool Dictionaries

Frequently Asked Question

4.29 • What are pool dictionaries?

Pool dictionaries are dictionaries which are examined by the compiler to resolve names found in methods. Each class can define its own pool dictionaries. Pool dictionaries are not inherited.

Pool dictionaries are usually used to hold the names of constants, often achieving the same purpose as header files of `#DEFINE` statements as used in C and C++.

Pool dictionary *names* are, however, themselves global variables and are stored in the `Smalltalk` dictionary.

Example 9 shows a possible pool dictionary definition.

```
" Definition of DiddleZork pool dictionary "
| pool |
pool := Dictionary new.
pool at: #FlagBits put: 2r10001000.
pool at: #ZonkFlag put: 16rFFFF0000.
pool at: #DiddleIt put: 2r00000001.
Smalltalk at: #DiddleZork put: dict
```

Example 9

In IBM Smalltalk, the kind of key used in the Smalltalk dictionary is not documented; instead, a message can be sent to a string to answer the correct key. Example 10 illustrates the technique:

```
" Definition of DiddleZork pool dictionary for IBM Smalltalk "
| pool |
pool := Dictionary new.
pool at: #FlagBits put: 2r10001000.
pool at: #ZonkFlag put: 16rFFFF0000.
pool at: #DiddleIt put: 2r00000001.
Smalltalk at: 'DiddleZork' asGlobalKey put: dict
```

Example 10

If a method uses the pool dictionary `DiddleZork`, it might hold code like that in Example 11.

```
" Use of DiddleZork pool dictionary "
zorkIt
   ^ FlagBits bitOr: DiddleIt
   " Answers: 89 "
```

Example 11

Pool dictionaries can be set as in Example 12, but it is bad practice. Pool dictionaries should only hold constant values.

```
" Setting values in DiddleZork pool dictionary "
badZork
   FlagBits := 2r10001111
```

Example 12

Frequently Asked Question

4.30 • What pool dictionaries come with Smalltalk?

Each vendor provides a different set of pool dictionaries.

IBM Smalltalk

Pool Dictionary	**Dictionary Contents**
`CfsConstants`	Constants used by file system calls
`CgConstants`	Constants used by graphics calls
`CldtConstants`	Characters often used in printable strings: `Tab`, `Cr`, ...
`CwConstants`	Constants used by widgets calls
`SystemExceptions`	Values used in exception handling

Digitalk V/Mac 2.0

Pool Dictionary	**Dictionary Contents**
`CharacterConstants`	Values of ASCII characters: `Tab`, `Space`, `Lf`, `Cr`, ...
`EventConstants`	Values like: `Button1`, `Button2`, `ShiftKey`, ...

ParcPlace VisualWorks 2.0

Pool Dictionary	**Dictionary Contents**
`IOConstants`	Values like: `CR` and `LF`
`TextConstants`	Values like: `CR, LF, Space, Tab, Ctrl`*n*
`OpcodePool`	Map opcode names to numeric values
`SymbolicPaintConstants`	Logical colors
`Undeclared`	Where ’undeclared’ variables live

VMARK Enfin Smalltalk

Pool Dictionary	**Dictionary Contents**
`AvailableColors`	Common colors and their values
`AvailableCurrency`	Country names and corresponding currency symbols
`AvailableCurrencyPositions`	Is currency symbol prefix or suffix?
`FileAccessConstants`	Constants used in file access calls
`FileOpenModeConstants`	Constants used in file opening
`PageSizeDict`	Paper sizes

Frequently Asked Question

4.31 • Should pool dictionary names use prefixes?

Yes, although most vendors do not do this for their own pools. If you define a pool dictionary, use a meaningful prefix to assist in identifying the values when others see them in the code.

For example, if your are defining a pool dictionary which holds various limits on financial transactions, you might name the pool dictionary `FinancialLimits` and prefix each entry with `FinLim`:

Pool Constant	**Description**
`FinLimWireTransfer`	The largest allowed wire transfer
`FinLimATMDailyWithdrawal`	The largest ATM cash withdrawal for a single day
`FinLimATMMinimumIncrement`	The smallest bill carried in an ATM machine

Frequently Asked Question

4.32 • Are there alternatives to pool dictionaries?

Yes.

Class methods

If there are not a huge number of values, then class methods can substitute for pool dictionaries. If the values apply to just one class (and its subclasses) and there are few of them, then the methods might go directly in the class itself. If there are more than a few, consider making a new class just to hold the constants.

Instance variable with a dictionary value

Rather than making a pool dictionary, make the same dictionary but put it into an instance variable. Each instance can share the same dictionary, or might have slightly different dictionaries, depending on the needs of the application. Access to the dictionary is simply:

```
myConstants at: 'FinLimWireTransfer'
```

Class variable with a dictionary value

Rather than making a pool dictionary, make the same dictionary but put it into an class variable. There is just one dictionary for the class and it takes up no space in the instance. Access to the dictionary is simply:

```
MyConstants at: 'FinLimWireTransfer'
```

Class instance variable with a dictionary value

Rather than making a pool dictionary, make the same dictionary but put it into an class instance variable, and make a class method that answers the dictionary. There is just one dictionary for the class and it takes up no space in the instance. Access to the dictionary is simply:

```
self class myConstants at: 'FinLimWireTransfer'
```

Frequently Asked Question

4.33 • When should pool dictionaries be used?

Pool dictionaries should be only when these four criteria hold true:

When there are a lot of named values

If there are just a few values it is better to have a class method return the value.

When the values go together in a coherent way

If the values are not connected they have no business in a pool dictionary together. They should be either in multiple pools or not in pools at all.

When the values are constant

Do not use pool dictionaries for values that change. Use them only for constant values, and write some code somewhere that initializes the pool dictionary when the application or class is loaded. Pools that are changed are giant traps for the unwary and will rise up and bite at inopportune times.

When the dictionary can be used by more than one class

If the values in a dictionary belong to just one class they are better off in an instance variable or class variable, even if though means having to do an explicit `at:` to get the value.

Classes

Frequently Asked Question

4.34 • How can I initialize classes?

Classes are initialized by code in the class itself. By convention, a class method named `initialize` is used.

Often the code is run once, by manual invocation, when the class is defined. When making changes to the dictionary, it is easy to forget to run the method. Many programmers put a comment at the top of the method:

```
" Don't forget to re-initialize me when you make changes:
       ThisClassName initialize     "
```

and then select the expression and evaluate it after making a change.

The `initialize` method can also be invoked by some other class method, as in Example 13. Note that the first `initialize` message is sent to the class and the second is sent to the new instance.

```
new
   self initialize.     "Initialize the class"
   ^ super new
      initialize        "Initialize the instance"
initialize
   " Initialize the class "
   ...
```

Example 13

IBM Smalltalk and ENVY

Some implementations of Smalltalk automatically invoke a class `initialize` method when the class is loaded from a fileout. (To be accurate, the fileout actually places code in the file which, when filed-in, will invoke `initialize`.) VMARK Enfin Smalltalk, IBM Smalltalk, and systems with Envy do this; others may. The fileout is generated with a line for each class initialize method like this:

```
Clunker initialize !
```

When classes are loaded from an unload file (using Envy or IBM Smalltalk), the class is sent the message `loaded`. It can then initialize itself.

> "ENVY/Developer has two ways of automatically initializing stuff on load: `loaded` and the `SubApplication>>toBeLoadedCode`. The `toBeLoadedCode` message is run *before* your app is loaded. It is primarily used for things like pool dictionaries which are required for code to be properly compiled or linked on load. Basically it is just an arbitrary string that you define to do whatever you want. On load, the string is fetched, compiled and run."
>
> "One thing that you should watch out for is multiple initializations. It is quite common to call, from `MyApp class>>loaded`, the initialization method of several classes `MyApp` defines. For initialization methods that initialize a class variable, you only want to initialize it once but if you blindly send #initialize (or whatever) to subclasses which inherit it from the class with the classVar, you could be initializing the classVar many times. Worse, you could reinitialize it when you load some app which subclasses your class."
>
> "I generally define two load initialize paths: `initializeOnLoad` and `initializeOnLoadOneTime`. The first I automatically send to all classes defined by the app. This is necessary to initialize class instance variables (i.e., you must initialize each class individually). The second I send only to defined classes

which directly implement `initializeOnLoadOneTime`. So, my `loaded` method looks something like:"

```
loaded
   self defined do: [:aClass |
      aClass initializeOnLoad.
         (aClass class includesSelector: #initializeOnLoadOneTime)
            ifTrue: [aClass initializeOnLoadOneTime]]
```

—Jeff McAffer, jeff@is.s.u-tokyo.ac.jp

Frequently Asked Question

4.35 • How do I create new classes from within a method?

The details of this differ; peek inside your favorite browser for details. But, note that this operation can only be done in the development environment; most vendors don't allow it in a packaged application.

Frequently Asked Question

4.36 • How do I add a method to a class from within a method?

The details of this differ; peek inside your favorite browser for details. But, note that this operation can only be done in the development environment; most vendors don't allow it in a packaged application.

Frequently Asked Question

4.37 • What is a Metaclass?

A metaclass is an instance of class `Metaclass` (or `MetaClass` in some systems). Each class is an instance of an instance of `Metaclass`.

See *12.13 'Where is the 'new' method (or: Why is 'new' THERE!)?'* for details.

Frequently Asked Question

4.38 • How should an object be checked for class membership?

It's a bad idea to check an instance to see what its class is. It often indicates some real problem with the design of a class when it seems necessary to check an instance for membership in some class.

If it is necessary to check an object for class membership, do *not* do this:

```
   thisThing class == Integer             "WRONG"
```

Instead, check for membership in a hierarchy by asking if an instance belongs to a class or one of its subclasses.

```
      thisThing isKindOf: Integer         "RIGHT"
```

Why? Because new subclasses can appear and disappear. The instance you have today may turn into an instance of a subclass tomorrow. Integers are a prime example. There never are any instances of `Integer` around. There may be instances of `SmallInteger`, `LargePositiveInteger`, and `LargeNegativeInteger`, or instances of `SmallInteger` and `LargeInteger`, or some other set of subclasses of `Integer`. Further, just what values map to just what classes varies by implementation.

This is true of most well designed hierarchies; classes come and go. Basic structure is less apt to change.

Frequently Asked Question

4.39 • Are there any tricks for checking for class membership?

One ’trick’ used by a number of standard classes is to add two methods, both named something like `isMyKindaClass`, one in the class `MyKindaClass`, which answers `true`, and one in `Object`, which answers `false`. The implementation of this for `Integer` looks like this:

```
   ! Object methods !
   isInteger
      ^ false

   ! Integer methods !
   isInteger
      ^ true
```

Example 14, Filein

The downside is obvious: you can add a lot of methods this way if you check for class membership a lot.

4.40 • What do Smalltalk variables hold?

It is sometimes said that Smalltalk has solved the ’pointer problem’ since Smalltalk appears to not have any pointers at all. Smalltalk has solved the ’pointer problem’, but it did it by making *everything* a pointer rather than eliminating them. Since everything is a pointer (and there are no pointer manipulation operations), Smalltalk does not have the exposures of, say, C or C++ to pointer abuse.

Every value in every variable in Smalltalk is a pointer to the value it represents†. However, rarely does this fact become visible in programs. Since all variables always hold pointers to objects, it is common to speak as if variables held the objects themselves. Rather than saying *the string referenced by* `name` it is common to say *the string in* `name`.

See [7.4] *’How big are SmallIntegers?’* on page 136.

See [5.6] *’When do pointers become visible?’* on page 90.

4.41 • What is a type?

Popular languages that most programmers are familiar with, such as C, C++, COBOL, FORTRAN, PL/I, and Pascal, have types. A variable, say `zot`, cannot be declared without specifying, explicitly, or implicitly by some default rules, what it 'holds'. This is done using language keywords, as in C:

```
long int zot;
float flot;
```

The variable `zot` holds (or describes or references or whatever) some memory. On many machines this will be 32 bits of memory aligned in some way. Since the type is `long int`, the compiler will generate code that treats these bits as an integer.

†. This is true even of integers although a universally implemented optimization encodes the value of smaller integers within the pointer itself. Such encoded integers are instances of `SmallInteger`.

The variable `flot` holds (or describes or references or whatever) some memory. This will be 32 bits of memory aligned in some way. Since the 'type' is `float`, the compiler will generate code that treats these bits as a floating point number.

The bits are otherwise the same. There is no distinction between memory locations that hold integers of 32 bits length and floating point numbers of 32 bits length. (Using some well-defined constructions, C programmers can even access the same memory 'cell' with variables of various types; FORTRAN programmers can, too, even more easily, with equivalence declarations.)

The language has to know at compile time what type a variable is, and it generates code using that knowledge. There are no tags on the data saying that 'this is an integer' or 'this is a float'.

Types are a characteristic of variables which aid the compiler in producing proper code.

Frequently Asked Question

4.42 • Does Smalltalk have types?

See [4.41] *'What is a type?'* for an earlier question on the same topic and [4.43] *'Is there any disagreement about types in Smalltalk?'* on page 71 for other views.

> As a noun, in programming, *type* defines the nature of a variable -- for example, integer, real number, text character, floating-point number, and so on.
>
> *—Microsoft Press Computer Dictionary. 2nd Ed.*

> Smalltalk ... is untyped.
>
> *—Cook, Hill & Canning, "Inheritance is not Subtyping" in Theoretical Aspects of Object-Oriented Programming, MIT Press, page 516.*

Smalltalk has variables but no language keywords to specify what the variable holds. Variables are declared by listing their names:

```
| zot flot |
```

The variables `zot` and `flot` can 'hold' objects.

See [4.40] *'What do Smalltalk variables hold?'*

So what are the types of `zot` and `flot`? There aren't any. (One could just as well say that they have the type 'object', but does a language with one type have types?)

Now, consider this code:

```
| zot flot |
zot  := 23.
flot := 23.45.
```

What are the types of the variables `zot` and `flot`? Does `zot` now have the type '`Integer`' and flot the type '`Float`'? No. They didn't have a type before and they don't after assignment.

Assigning a value to a variable in Smalltalk does not change the type of the variable, at least without doing great damage to the popularly held view of what 'type' means. A variable may ’hold’ an integer without being of type `Integer`.

Now, consider:

```
| zot flot |
zot  := 23.
flot := 23.45.
flot := flot asInteger
```

Has `flot` changed its type? No!

Consider this code:

```
holdsNil: aCollection
   aCollection
      do: [ :element |
         element = nil
            ifTrue: [ ^ true ] ].
   ^ false
```

What is the type of `aCollection`? Does it change its 'type' every time that the method is called? In this method `aCollection` can be *any* object that responds to `do:` by invoking a block and passing one parameter to it. It doesn't have to be a collection.

Saying that all types derive from type `Object`, as some proponents of other languages do, misses the point entirely. There are no types to derive from. `Object` is no more a type than `Float`.

Smalltalk inheritance is not a type inheritance but an implementation sharing inheritance. A subclass is not a subtype but an implementation of a new object that shares some or all of the implementation of the parent class.

Smalltalk values are tagged. The value in the variable `aCollection` is self describing. There is no need to specify that a variable holds a collection. The compiler can generate code without having any type information -- the only operations in Smalltalk are message sends and assigns.

Smalltalk compilers optimize generated code, sometimes making non-binding assumptions about what a variable might hold. See [12.3] *'How can Smalltalk be optimized?'*

Frequently Asked Question

4.43 • Is there any disagreement about types in Smalltalk?

There is some disagreement on this topic, to put it mildly; saying that it causes flame wars again puts it mildly.

Some claim that, while it is literally true that Smalltalk variables have no type declaration, classes are really the types in Smalltalk programs. This seems to redefine what the term *type* means from a characteristic of a variable to a characteristic of data. While this kind of redefinition happens across time with all human languages, it is an especially dangerous thing to happen to technical terms which must have a precise definition in order to be useful at all.

Others argue that Smalltalk *should* have a type system, and that programmers should declare the types of variables. OK, but then the result is not Smalltalk, it is some new language similar to Smalltalk. It may be better, or it may not, but it's not Smalltalk.

Yet others argue that some new thing should be introduced to replace types. In one proposal, the type of a variable becomes a list of the classes whose instances the variable might at some time hold. Again, this really defines a new language possibly similar to Smalltalk.

(Need to get pointers into the literature.)◄

Blocks

Frequently Asked Question

4.44 • What are blocks?

Blocks are expressions or groups of expressions enclosed in square brackets:

```
[ 2 + 3 ]
```

```
[  a := 1.
   b := 2.
   c := a + b ]
```

A block is itself an expression; its value is the block itself, not the result of evaluating the contents.

Blocks resemble small subroutines:

- Blocks must be explicitly invoked. Example 15 answers 1:

  ```
  | a |
  a := 1.
  [ a := a + 1 ].
  ^ a
  " Answers: 1 "
  ```

  Example 15, Display

  The block in the third line is effectively ignored; its value is the block itself but nothing is done with the result. The block must be explicitly invoked, or evaluated, as in Example 16:

  ```
  | a |
  a := 1.
  [ a := a + 1 ] value.
  ^ a
  " Answers: 2 "
  ```

  Example 16, Display

  Example 16 answers 2.

- Blocks can have parameters. Parameters are described in a header in the block; each starts with a colon:

  ```
  [ :parameter1 :parameter2 … |
     expression … ]
  ```

  Parameterized blocks are invoked with one of the *value* messages: `value:`, `value:value:`, `value:value:value:`, or `valueWithArguments:`.

  Example 17 answers 2:

```
| a |
a := 1.
[ :increment |
   a := a + increment ] value: 1.
^ a
" Answers: 2 "
```

Example 17, Display

- Blocks can have local variables (in most implementations, but see [4.45]). Local variables follow the parameter header and have the same format as local variables in a method: they are a list of names surrounded by vertical bars.

```
[ :parameter1 :parameter2 … |
   | localVariable1 localVariable2 … |
   expression … ]
```

Example 18 answers 2:

```
| a |
a := 1.
[ :increment |
   | temp |
   temp := a + increment.
   a := temp ] value: 1.
^ a
" Answers: 2 "
```

Example 18, Display

See [4.49] *'How about optimizations of blocks not simply executed inline?'* on page 75 for information on the performance impact of block local variables.

Frequently Asked Question

4.45 • Are blocks objects?

Blocks are full fledged objects. They can be assigned to variables, placed into objects, and passed as parameters. The sort block used by sorted collections is a good example of passing a block as a parameter.

```
SortedCollection sortBlock: [ :first :second | first < second ]
```

Sort blocks can be assigned to variables and used later:

```
descending := [ :first :second | first > second ].
ascending  := [ :first :second | first < second ].
...
SortedCollection sortBlock: ascending
```

4.46 • When are block objects created?

Block objects are created at the point in a program when they are invoked, passed as a parameter, or assigned to a variable.

In Example 19, the block refers to a method local variable, a. Changes to the variable are reflected in the block no matter when it is evaluated.

```
| a block |
a := 1.
block := [ a ].
a := 3.
block value
" Answers: 3 "
```

Example 19, Display

In Example 20, the outer block is passed the block local variable, a, and it answers another block which answers the value of the parameter. This binds the value of x to that of a at the time the inner block is answered.

```
| a block |
a := 1.
block := [ :x | [x] ] value: a.
a := 3.
block value
" Answers: 1 "
```

Example 20, Display

In some cases, the compiler can optimize away the need to actually create a block object. In other cases, the block object must be created.

See also Example 23 in 4.50 Do blocks differ between implementations?

4.47 • When is it useful to put blocks in variables?

Many authorities think that blocks should rarely be put into variables. But there are times when it is a useful practice, especially in just those circumstances where a large switch statement in C is useful: given some integer value with a large number of possible values, choose some action that depends on the value.

Need a good, short example◄

Frequently Asked Question

4.48 • Are blocks optimized?

Blocks can be optimized and most implementations do some degree of optimization of blocks.

In the simplest of cases, `ifTrue:` and `ifFalse:` are not sent, and the block's code is expanded inline. While this may require compiling code for two cases (when the receiver *is* and *is not* boolean), it is much faster in the common case.

Frequently Asked Question

4.49 • How about optimizations of blocks not simply executed inline?

Optimizations can be performed on blocks that will be passed as parameters or stored into variables. These optimizations can make a considerable difference in both execution time and memory use.

In the general case, a block has to know about the method context in which it was assigned or from which it was passed. The block might refer to parameters of the method, to local variables of the method, or to instance variables of the object, or it might contain a return statement that would cause a return from the method.

If a block contains no references to variables outside of itself and has no return statement, then the block object can be smaller and simpler. A block that references only instance variables might also have a block object that is simpler than the general case.

In order to allow an implementation to perform those optimizations it can perform, it is best to code blocks according to these guidelines:

Make all temporary variables local to the block

Do not define *outside* the block those temporary variables that will be used only *inside* the block. That is, write:

```
| x |
x := [ :parm |
   | local |
   ... local ... ]
```

instead of:

```
   | x **local** |
   x := [ :parm |
      ... local ... ]
```

Pass variables instead of referring to them directly

Instead of referencing variables directly, pass their values as parameters. That is, write:

```
   | x |
   x := [ :parm |
      ... parm ... ].
   x value: instVar
```

instead of:

```
   | x |
   x := [ ... instVar ... ].
   x value
```

Avoid returns from stored or passed blocks

It's generally a bad idea to have a return from a block that is stored or passed since there is no guarantee that the context in which the block object was created still exists. Such blocks cannot be optimized since they must carry along the whole context of the method invocation in the method where they are written.

Frequently Asked Question

4.50 • Do blocks differ between implementations?

Yes. In particular, many Digitalk versions of Smalltalk, and earlier versions of VisualWorks do not support local variables within blocks; all locals variables must be declared at the method level. Worse, they consider block parameters as method-wide local variables.

For example, Example 21 may have two method-wide local variables (`counter` and `index`) or just one (`counter`), depending on which implementation is used:

```
| counter |
counter := 0.
(1 to: 20)
   do: [ :index |
      counter := counter + index ].
```

Example 21, Display

Example 22 may answer some integer (in this case probably `2`), instead of the collection, when run on a Smalltalk with method-level local variables.

```
| a b |
a := #(9 1 6 4 8).
b := SortedCollection sortBlock: [ :a :b | a > b ].
b addAll: a.
^ a
```

Example 22, Display

The problem is that the parameters have the same names as the two declared local variables, and the values passed to the block when it is sorted by the `addAll:` replace the values explicitly assigned. Other systems should flag such uses as errors.

Example 23 assigns blocks to `blockArray`.

```
| blockArray dataArray |
blockArray := Array new: 5.
dataArray := #( ’Apple’ ’Orange’ ’Grape’ ’Lemon’ ’Kiwi’).
1 to: blockArray size do: [ :index |
   blockArray
      at: index
      put: [ dataArray at: index ] ].
^ (blockArray at: 2) value
```

Example 23, Display

The block:

```
[ dataArray at: index ]
```

has a reference to the variable `index`. If `index` is a method-level local variable, then it is the same variable for each block stored and the desired effect will not occur.

If `index` is a block-local variable, then it is a different variable each time the block is invoked, and a different value will be stored with the block; this is the desired situation.

The example ends with the return of the value of the second saved block.

When run on Digitalk Smalltalk/V-Mac 2.0, Example 23 gets a walkback in the stored block because of an index out of bounds. The value of `index` is 6, indicating that it is one more than the last value it had in the loop; this is the value which caused the loop to terminate.

When run on ParcPlace VisualWorks 2.0, this code answers the string `’Orange’`. The value of `index` must be 2; this is the value it had when the block was stored.

—Adapted from Ralph Johnson, 'Classic Smalltalk Bugs'

Frequently Asked Question

4.51 • What is a closure?

A *closure* is a *closed expression*, or an expression which carries along the meanings of its free variables. Closure is not a Smalltalk term, but comes from language theory. In Smalltalk, blocks in implementations which have local state (local variables private to the block) are called closures. Blocks in implementations where there are no block local variables and/or where block parameters are local variables in the containing method are not closures.

Check in Lisp books; refs.◄

Methods

Frequently Asked Question

4.52 • What are private methods?

Private methods are methods whose author marked them as private using some convention or another. The idea is that the author of the code will not himself use them from outside the class tree in which they are defined. The author of the code has hopes that his friends won't use them either.

Private methods are indicated different ways in different implementations:

Digitalk Smalltalk

Indicated by a comment at the front of the method.

(What happens with Team/Tools?)◄

IBM Smalltalk and systems with ENVY

A separate method type is maintained by the system. Browsers can show public, private, or both.

ParcPlace VisualWorks

Separate categories are created for private methods.

VMARK Enfin Smalltalk

Private methods start with the word ’`private`’.

Frequently Asked Question

4.53 • Can the privacy of private methods be enforced?

No. Marking a method private is like telling your dog not to eat that steak you're leaving out on the floor. Maybe your dog is well trained...

There are ways to see if a private method is being used from outside its intended scope. The techniques vary slightly by implementation.

VisualWorks

```
aPrivateMethod
   thisContext sender receiver == self
      ifFalse: [^self error: 'Private method'].
   " do private things "
```

This technique is often too expensive for frequent use. Using conditional compilation techniques described in a January 96 column in *The Smalltalk Report* by Jan Steinman and Barbara Yates , you could do such privacy checks during testing, and remove them for delivery.

—Suggested by: Jan Steinman, Barbara Yates <barbara.bytesmiths@acm.org>

IBM Smalltalk

The heart of the technique is a one-liner which answers the caller of the current method.

```
Processor activeProcess stackAtFrame: 1 offset: -3
```

If used from within a block, the results are undefined; it usually will answer the method that invoked the block but that’s not the same as the method that invoked the method that contains the block.

It’s handy to package up such expressions, say by making a new method. The first bit of code below is a new method for `Object` which answers its callers caller. (Thus the ’`1`’ becomes a ’`2`’ indicating one level further up the stack.)

```
! Object methods !
sender
   ^ Processor activeProcess stackAtFrame: 2 offset: -3
```

Example 24, Filein

This method can be used to check for privacy:

```
myMethod
   self sender = self
      ifFalse: [ self error: 'Private method' ].
   " ... do your stuff ... "
```

A more comprehensive method, again for `Object`, is a bit slower but is simpler to use. It performs the full check for privacy and issues a message if violated:

```
! Object methods !
isPrivateMethod
   | activeProcess sender sendersSender |
   activeProcess := Processor activeProcess.
   sender        := activeProcess stackAtFrame: 1 offset: -3.
   sendersSender := activeProcess stackAtFrame: 2 offset: -3.
   sender = sendersSender
      ifFalse: [ self error: 'My sender is not myself.' ]
```

Example 25, Filein

Use it this way:

```
myMethod
   self isPrivateMethod.
   " ... do your stuff ... "
```

After testing is completed, the code can be removed, or the definition of `isPrivateMethod` can be changed to do nothing.

Inheritance, Self and Super

Frequently Asked Question

4.54 • What is self?

The special variable `self` represents the object on behalf of which a method is being run. It may refer to an instance of the class in which the method is located, or it may be an instance of a subclass.

The value of `self` can be considered a hidden parameter in all message sends. For example, in the expression:

```
employee name: aString
```

two parameters are passed: first, the value in `employee` which will become `self`, and then the value in `aString`.

Messages sent to `self` are sent to the object that `self` represents.

Frequently Asked Question

4.55 • What is super?

The special variable `super` is another name for `self`. It has one special property: messages sent to `super` are bound to a method found by starting the search in the parent of the class in which the method is written.

In its simplest and most common case, `super` is used in user written class methods named `new`.

```
" The standard new method idiom "
new
   ^ super new
      initialize
```

Example 26

In Example 26, an instance needs to be initialized. The message `super new` gets a new instance, sends the `initialize` message to it, and returns the new instance (assuming that `initialize` answers `self`, which it usually does).

The expression `self new` could not have be used since it would recursively reinvoke the method. Sending `new` to `super` solves the problem by asking the parent to do the work.

See [4.57] *'When should super not be used?'*

Frequently Asked Question

4.56 • Why can't I pass super as a parameter?

The `value` of `super` is identical to the value of `self`. If `super` is passed as a parameter or assigned to a variable, the value passed or assigned is that of `self`. The special properties are lost.†

Messages intended for the parent of the class must be sent directly to `super`, literally:

```
super aSelector
```

Frequently Asked Question

4.57 • When should super not be used?

The variable `super` should *only* be used when the message name is the same as the current method name. That is, it should always be true that `methodName` is the same as `messageName` in:

```
methodName
   super messageName
```

Messages sent to any other name should use `self` and not `super`. Using `super` bypasses method(s) that belong to `self` (or lower parents of `self`). If that is your intent, you need to carefully rethink what you are doing. Such code is ugly, tricky, hard to read, hard to maintain, and will rise up and zap you or a teammate later.

Note to implementers: Compilers really should warn about this case.

Example 27 is similar to code found by the author in a commercial product. The `fromFile:` method uses `super new` to bypass the unwanted `buildTitle` in `new`.

†. The draft ANSI Smalltalk standard allows `super` to be used only as a target of a message send.

```
" fromFile: problem (class methods) "
new
   super new
      buildTitle
fromFile: aString
   super new
      fileName: aString;
      buildTitleFrom: aString
```
Example 27

The problem comes when subclassing this code and overriding `new`. The method `fromFile:` will then bypass the subclasses' `new` method. Such bugs are no fun to find. The '`super`' in the `fromFile:` method should be '`self`'. While it does build the title more than once, the code can also be subclassed.

If it is not feasable to build the title more than once, a different approach must be taken. If `fromFile:` is invoked it builds a title. If not, then it is built the first time it is needed.

```
" fromFile: 'solution' (class methods) "
fromFile: aString
   self new
      fileName: aString;
      buildTitleFrom: aString

" (instance methods) "
methodThatUsesTitle
   ...
   title := self title.
   ...
buildTitleFrom: aString
   title := aString, ...
buildTitle
   title := ...
title
   title isNil
      ifTrue: [
         self buildTitle ].
   ^ title
```
Example 28

This is an example of lazy evaulation. See "How can I defer calculations until they are needed?" in chapter "Tips and Tricks" on page 154.

4.58 • What is the difference between self and yourself?

The word `self` is a reserved word in Smalltalk that refers to the object which is the receiver of a message.

The word `yourself` is a message name which can be sent to any object. In response, the object answers `self`. That is, `yourself` has the implementation:

```
yourself
   ^ self
```

The message `yourself` is used in cascaded message sequences to assure that the value answered by the cascade is the receiver of the cascade. For example, the code:

```
oc := OrderedCollection new
   add: 'hello';
   add: 'there'
```

Example 29, Display

assigns `'there'` to `oc` since the last `add:` returns '`there`'. However, the code:

```
oc := OrderedCollection new
   add: 'hello';
   add: 'there';
   yourself
```

Example 30, Display

assigns the new instance of `OrderedCollection` to `oc` since `yourself` always answers the receiver.

—David Buck, The Object People

(Need to ref questions on add:.)◄

4.59 • Are there any uses of yourself except in cascades?

"Cascaded messages account for virtually all of the cases where `yourself` is used. There are some other possible uses, but they are contrived. Suppose I maintained a dictionary of messages to send to different objects to get back a string to display on the screen. I could write:

```
stringFor: anObject
   ^anObject perform: (dictionary at: anObject class)
```

In the dictionary, I could have:

```
       Integer->#printString
       Client->#formattedPrintString
       String->#yourself
       ...
```

I would think twice (maybe three times) about using such a technique. It does, however, demonstrate that it's possible to have another meaningful use of `yourself`."

—David Buck, dbuck@ccs.carleton.ca

VisualWorks has an unusual use of `yourself` in the class `LensAbstractProxy` which contains the code:

```
   self yourself instVarAt: anIndex
```

But `LensAbstractProxy` is not a subclass of `Object` and there is game playing with `doesNotUnderstand:` going on under the covers. Take a look if you like puzzles.

—Thanks to John McIntosh

Part V: Objects

This chapter covers both the protocol of class `Object` and information on objects in general.

Object Identity

Frequently Asked Question

5.1 • What is equality?

Objects are *equal* when they have the same value. The #= message compares two objects for equality. However, what it means to ’have the same value’ depends on the object. Generally, two objects have the same value when their parts are equal.

Equality testing can be slow, depending on the objects. A comparison of two strings may compare each character in one string with the corresponding character in the other string. A comparison of two arrays may, depending on the vendor, compare each element of the two arrays, recursively.

Frequently Asked Question

5.2 • What is identity?

Objects are identical when they are the same object. The #== message compares two objects for identity.

Identity testing is fast since it is necessary only to compare two object pointers rather than the data to which the pointers refer.

Identity is not always obvious and it is easy to type == instead of = even when you know the difference.

A long discussion was held about Example 31 on `comp.lang.smalltalk` in the Spring of 1996:

```
(1/5) == (1/5)
```
Example 31

A poster was puzzled about why Example 31 answers `false` when Example 32 answers `true`:

```
5 == 5
```
Example 32

The explanation is simple. The expression `(1/5)` is an expression and is evaluated at run time. The two identical expressions are each evaluated and each answer different objects.† Thus the == method is asnwering correctly that the two objects are not the same object.

Example 32, on the other hand answers `true` because the integer `5` is an identity object and equal identity objects always identical. (See question [5.3].)

More confusing is Example 33.

```
5.0 == 5.0
```
Example 33

The answer depends on the implementation; some answer `true` and some answer `false`. The reason is less obvious. Some compilers keep a list of literals that have to be generated in the resulting code. When a literal is encountered it is compared with those already in the list; if present, the one in the list is reused. Thus, each reference to a literal in a method with a given value refers to the same object and the expression in Example 33 answers `true`. Other compilers might emit each literal when found causing Example 33 to answer `false`.

Frequently Asked Question

5.3 • What are identity objects?

Identity objects are always identical, that is, are the same object, when they have the same value. Small integers are examples of identity objects. Each value of, say, `12`, is the same object as all other integer `12`'s.

Identity objects always compare `true` with #== when they compare `true` with #=. Since #= answers `true` when the values are the same, and #== answers `true` when

†. Why won't a Smalltalk compiler evaluate `(1/5)` at compile time and eliminate the duplicate runtime evaluation? It's because the definition of integer divide might change, either due to a bug fix, or to insert debugging code, or for other reasons. In general, there is no simply way to perform operations on constants at compile time in a development environment since everything must be considered in a state of flux.

DRAFT

the two objects are the same, operations that produce identity objects must always produce the same object for a given value no matter how derived or calculated. Adding `2` and `2` always produces the same `4` object and thus `(2+2)==4` is always `true`.

Identity objects include small integers, characters, symbols, and `true` and `false`.

```
Always true                      Always true
7 = 7                            7 == 7
(1+4) = 5                        (1+4) == 5
$b = $B asLowercase              $b == $B asLowercase
true = true                      true == true
#xy = ’xy’ asSymbol              #xy == ’xy’ asSymbol
```

But:

```
Always true                      Always false
(2.5 + 1.5) = 4.0                (2.5 + 1.5) == 4.0
(’ab’, ’xy’) = ’abxy’            (’ab’, ’xy’) == ’abxd’
(1@1) + 5 = (6@6)                (1@1) + 5 == (6@6)
```

Frequently Asked Question

5.4 • What are immutable objects?

Immutable objects cannot be modified internally. All operations on immutable objects that answer a new value also answer a different instance. Most objects are not immutable. Immutable objects include all numbers, characters, and symbols. Identity objects are always immutable, but the converse is not true.

Copies of immutable objects that are not identity objects always return a new instance when a method causes the object to have a new internal value.

Copies of identity objects are always the same object as the original.

```
Always true                      Always true
2 copy = 2                       2 copy == 2
$a copy = $a                     $a copy == $a
true copy = true                 true copy == true
```

But, copies of other immutable objects are never the same object as the original.†

†. Some implementors don’t get this quite right; at least one commercial implementation answers `true` to `1.2 copy == 1.2`.

Always `true`	**Always `false`**
`(2@3) copy = (2@3)`	`(2@3) copy == (2@3)`
`'abc' copy = 'abc'`	`'abc' copy == 'abc'`
`1.2 copy = 1.2`	`1.2 copy == 1.2`

Pointers and Copying

Frequently Asked Question

5.5 • Does Smalltalk have pointers?

Yes. All variables hold pointers to objects, rather than the objects themselves. There is, however, no way to set the value of a pointer other than by instanciating a new object, and no way to explicitly dereference a pointer.

Frequently Asked Question

5.6 • When do pointers become visible?

Pointers are usually 'invisible' and people often talk as if variables hold objects rather than pointers to objects. Pointers become 'visible' and cannnot be ignored in some circumstances where several variables or collection elements are set to the same value.

For example, a common coding error by a beginning Smalltalk programmer involves initialization of an array of objects. In Example 34, an array is initialized in a loop with the value of a rectangle. Each loop sets a different value into the rectangle and then stores it into the array. The intent is to have five different rectangles with five different values.

```
" The pointer bug "
| array rect |
array := Array new: 5.
rect := 0@0 corner: 8@9.
1 to: array size do: [ :item |
   rect origin: item@item.
   array at: item put: rect ].
^ array
" Answers: #(5@5 corner: 8@9 5@5 corner: 8@9 5@5 corner: 8@9 5@5
corner: 8@9 5@5 corner: 8@9) "
```

Example 34, Display

Note that the result is five identical values, not five rectangles with different values since the same object is assigned to each element of the array.

The problem here is that there is but one rectangle; each array element holds a pointer to that one rectangle. When the origin of the rectangle is changed it is, of course, seen through all of its pointers.

The problem can be corrected several ways. First, in Example 35, a copy is made of the rectangle before it is put into the array.

```
" Fixing the pointer bug; first method "
| array rect |
array := Array new: 5.
rect := 0@0 corner: 8@9.
1 to: array size do: [ :item |
   rect origin: item@item.
   array at: item put: rect copy ].
^ array
" Answers: #(1@1 corner: 8@9 2@2 corner: 8@9 3@3 corner: 8@9 4@4
corner: 8@9 5@5 corner: 8@9) "
```

Example 35, Display

Second, and better, in Example 36, a new rectangle is created and put into the array.

```
" Fixing the pointer bug; second method "
| array |
array := Array new: 5.
1 to: array size do: [ :item |
   | rect |
   rect := item@item corner: 8@9.
   array at: item put: rect ].
^ array
" Answers:  #(1@1 corner: 8@9 2@2 corner: 8@9 3@3 corner: 8@9
4@4 corner: 8@9 5@5 corner: 8@9) "
```

Example 36, Display

Frequently Asked Question

5.7 • How should objects be copied?

There are several ways to obtain copies of objects. First, is to simply send the `copy` message to the object, as in Example 37.

```
" Copying a rectangle "
| rect1 rect2 |
rect1 := 1@1 extent: 4@4.
rect2 := rect1 copy
```
Example 37, Display

Second, is to allocate a new object with values from another, as in Example 38.

```
" Allocating a rectangle "
| rect1 rect2 |
rect1 := 1@1 extent: 4@4.
rect2 := rect1 origin extent: rect1 extent
```
Example 38, Display

Within a class, it is common to use the technique shown in Example 39, where the class is asked for a new instance and then values are filled in from the current instance.

```
" Copying self "
copy
   | aCopy |
   aCopy := self class new.
   aCopy value1: self value1.
   " ... etc ... "
   ^ aCopy
```
Example 39

The default `copy` method is certainly simpler, but it is not always faster. If making new instances fast is important, try it both ways in a test case first.

Frequently Asked Question

5.8 • How does the copy method make copies?

The `copy` method makes a copy of the instance and assigns the same object pointers to each instance variable. See Figure 2.

See *5.10 'When should deepCopy be used?'* for more information.

Frequently Asked Question

5.9 • What is deep copy?

A deep copy is a copy of an object in which some or all of the contents are also copied. The exact definition is vendor specific, with at least one vendor not providing a (public) implementation. See Figure 3.

See *5.10 'When should deepCopy be used?'* for more information.

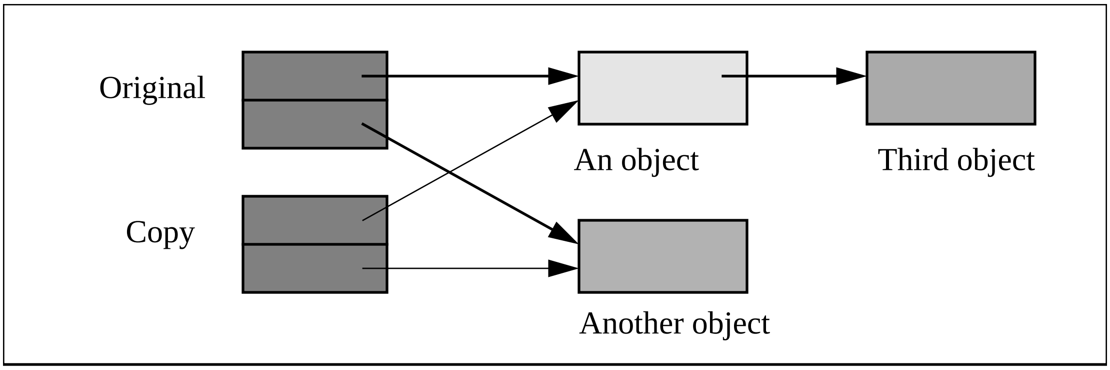

Figure 2: Copying Objects with `copy`. The copy of an object refers to the same contents as the original object.

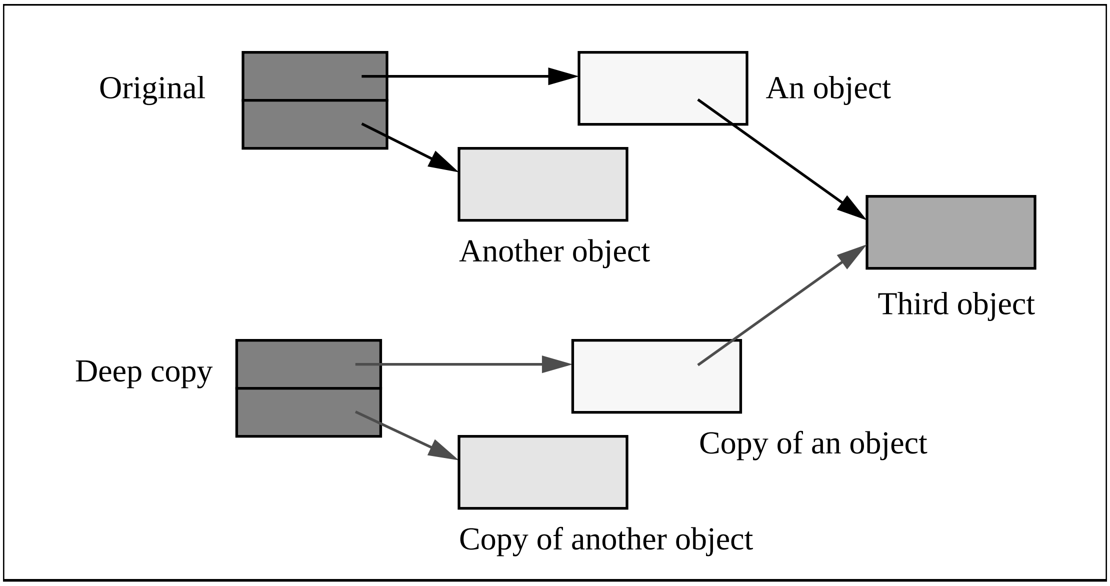

Figure 3: Copying Objects with `deepCopy`. The deep copy of an object refers to copies of the contents as the original object.

5.10 • When should deepCopy be used?

There are several definitions for what `deepCopy` means. Let’s define some terms. The first three (copy, shallow copy, and deep copy) are standard Smalltalk terms; the last two (full copy and true copy) are specific to this discussion.

copy

A shallow copy.

shallowCopy

A new copy of the instance but with instance variables pointing to the same objects. See Figure 2.

deepCopy

A new copy of the instance filled with shallow copies of the instance variables. Some Smalltalk systems implement `deepCopy` this way. (IBM Smalltalk has a private `deepCopy` that does this; VisualWorks does not seem to have a `deepCopy` method.) See Figure 3.

fullCopy

A copy of the instance filled with a `fullCopy` of each of the instance variables. This is what some people mean by deep copy, or 'fully recursive deep copy'. In general, a `fullCopy` might run forever and take infinite memory. It takes some thought and care not to do that.

trueCopy

What the customer really wanted. (Remember the drawings of the swings?)

In general, what it means to copy an object in Smalltalk depends on the object being copied. No generic shallow or deep copy methods will ever 'do the right thing' for all objects.

For example, let's take an object which has three instance variables:

`someInts`	A collection of integers
`someStuff`	A collection of other stuff
`anotherMe`	A pointer to another instance of the same class

A `trueCopy` method might be:

```
" Make a true copy of us "
trueCopy
   | copy stuff |

   copy := self class new.          "Get a new instance"

   copy someInts: someInts copy.    "Copy the integer collection"

   stuff := someStuff class new:    "Get new 'stuff' collection"
                  someStuff size.
   1 to: someStuff size do: [ :n |  "Copy each element"
      stuff at: n put:
         (someStuff at: n) copy ].  "(Copy does what it does)"
   copy someStuff: stuff.           "Store in new instance"

   copy anotherMe: anotherMe.       "Make no copy of anotherMe"
   ^ copy
```

Example 40

That is, the true copy of the integer array is a new array with the same objects. The true copy of the array of other stuff is a new array with copies of the objects; this is often just what a default `deepCopy` does. Note that the `copy` sent to the object can invoke an arbitrary copy routine, not necessarily the default one in `Object`.

The true copy of the pointer to `anotherMe` is no copy at all! This field is a link to another instance and we don't want to copy that object, just point to it.

Assuming that `deepCopy` answers a copy of the collection with a shallow copy of the contents, the code can be much simplfied:

```
" Make a true copy of us IF deepCopy works as noted "
trueCopy
   | copy stuff |
   copy := self class new.
   copy someInts: someInts copy.
   copy someStuff: someStuff deepCopy.
   copy anotherMe: anotherMe.
   ^ copy
```

Example 41

Now, what should the `copy` and `deepCopy` methods do for this object? Both should be invocations of `trueCopy:`.

```
copy
    "Answer a trueCopy"
    ^ self trueCopy
deepCopy
    "Answer a trueCopy"
    ^ self trueCopy
```

No system defined `deepCopy` method *can* do it right. Always code your own. A good side effect of doing your own is that you know what is really happenning!

It is very important to stop and *think* about what it means to copy an object. If there is a general purpose `deepCopy` then it will get used, people may be happy for a while, and then much later in the development process things will be observed to fail in strange ways. Much better to think it out up front.

Dependents and Dependencies

Frequently Asked Question

5.11 • What is a dependent?

A dependent is an object that is notified by another object (the object it is dependent upon) when something interesting happens. An object becomes a dependent of another when so specified by the `addDependent:` message and stops being a dependent when so unspecified by the `release` or `removeDependent:` messages.

Dependents are notified of a change by the `broadcast`, `broadcast:with:`, `changed`, and `changed:` messages.

Any object can be a dependent, but dependencies are usually used only in user interfaces.

Dependents provide a way to isolate objects that change from the objects that depend on the change. Without dependents, or a similar mechanism, each object that depends on another would have to be directly known to that other object.

Frequently Asked Question

5.12 • Does anyone really use dependencies?

Yes. Many graphical user interface systems use them internally and many users use them in conjunction with user interfaces to their applications. Uses in other contexts is rare.

Frequently Asked Question

5.13 • Is there a short, neat, example of dependents?

Good question!

Questions about become:

Frequently Asked Question

5.14 • What does the `become:` message do?

In theory, the `become:` message swaps the definitions of two objects. All pointers to one object are swapped for pointers to another.

In Example 42, an array is created and its pointer is placed into two variables, `a1` and `a2`, and a bag is created and its pointer is placed into `b1` and `b2`. Finally, a `become:` message, in theory, swaps the definitions of the objects pointed to by `a2` and `b2`.

```
| a1 a2 b1 b2 |
a1 := a2 := Array with: ’cat’ with: ’dog’.
b1 := b2 := Bag with: ’house’ with: ’home’.
a2 become: b2
```

Example 42, Display

In practice, there are two ways to implement `become:`.

Symmetric

The `become:` in the last line of Example 42 swaps the definitions of `a2` and `b2`; all pointers to the object pointed to by `a2` then point to the object pointed to by `b2`, and all pointers to the object pointed to by `b2` then point to the object pointed to by `a2`.

Since `a1` pointed to the same object as `a2` before the become:, it will points to the same object afterwards, and the same holds true for `b1` and `b2`. Thus all pointers to the objects are effectively swapped.

This is the original definition of `become:`.

One-way

The `become:` in the last line of Example 42 changes the definition of `a2`; all pointers to the object pointed to by `a2` then point to the object pointed to by `b2`. The definition of `b2` is unchanged.

Since `a1` pointed to the same object as `a2` before the `become:`, it will point to the same object afterwards; it is, in effect, changed to point to `b2`. The objects pointed to by `b1` and `b2` are the same, and are unchanged by the `become:`. The array object that `a1` and `a2` did point to are no longer pointed to and become garbage.

This difference has come about because of optimizations done in some modern implementations.

See *5.15 'How does become: differ between implementations?'* for more information.

Frequently Asked Question

5.15 • How does `become:` differ between implementations?

In olden days, Smalltalk implementations had an object table which held the real pointers to objects. Object pointers were indexes into the table. A `become:` was simply a swap of two pointers in the object table. It was fast, and symmetric.

But, object tables were of fixed size and one had to know ahead of time how big a table to allocate. Further, each object reference was indirect through the table: calculate the table address given the index, and then fetch the real pointer to the object, which causes an extra memory reference for each object reference.

So what to do? Without an object table, doing a full fledged `become:` requires scanning all objects in the world to find pointers to other objects and changing the affected pointers each place they are found. What was a few microseconds suddenly became several seconds. Not good!

So, there developed two schools of wizards. One school forbade symmetric `become:`s entirely. All `become:`s had to be asymmetric, with the second object simply replacing the first. While this still required scanning the world, it was faster than the work required to to a full symmetric `become:`.

The second school felt that `become:` was too important to go away. They made all object pointers indirect, not through an object table, but through a hidden pointer. While this took more memory, and might make object references slower that the other school's wizards' solution, it was better tha using an object table.

Which implementation uses which solution?

Implementation	**Technique**
IBM Smalltalk	Asymmetric
VisualWorks	Symmetric
Digitalk	Asymmetric
VMARK Enfin Smalltalk	?
GemStone Smalltalk	?

Frequently Asked Question

5.16 • Is there a test which tests which kind of become: is implementated?

Try this:

```
" Determine which kind of become: "
| a b |
a := String with: $a.
b := String with: $b.
a become: b.
a == b
   ifTrue:  [ ^ 'Asymmetric' ]
   ifFalse: [ ^ 'Symmetric' ]
```

Example 43, Evaluate

Frequently Asked Question

5.17 • When should I use `become:`?

The `become:` message is useful under a few circumstances. It should be used with caution since it can do great damage if improperly used.

See *5.18 'What are the pitfalls of using become:?'*

The `become:` message will be useful when:

- You need to destroy all references to an object.

Sometimes some other object retains pointers to your objects and, because of a bug, there is no way for that other object to finish processing and release your objects. This often happens during development; in some implementations a partly opened window that gets a walkback will leave things in such a confused state.

One solution is code like this:

```
" Replace all instances of MyObject with a new string "
MyObject allInstances
   do: [ :inst |
      inst become: String new ]
```

Example 44, Display

This code replaces pointers to each instance of `MyObject` with a pointer to a new string. Since there are no existing pointers to the new string, either the symmetric or one-way versions of `become:` will work.

See [5.19] *'How do I find and remove 'lost' instances?'* on page 101.

- You need to replace all references to an object with another.

Some object needs to be updated and you don't have control of all pointers to it. In some Smalltalk implementations, a collection that grows will replace all of the old copies of itself with new ones using `become:`.

```
oldVersion become: newVersion
```

This code replaces pointers to each instance of `oldVersion` with a pointer to `newVersion`. While there are existing pointers to the new version, they do not need to be swapped with the old version since the old version needs to become garbage. Either the symmetric or one-way versions of `become:` will work in this case.

There is an alternative implementation in which a collection object does not directly contain the collection elements, but has an instance variable which contains another collection which holds the elements. In this situtation the only pointer to the elements is known and can easily be replaced with a new element collection without using `become:`.

- You need to swap all references to an object with another object.

(Are there any good reasons for this??)

—Adapted from Ralph Johnson's Classic Smalltalk Bugs

5.18 • What are the pitfalls of using `become:`?

The `become:` message is dangerous since it is easy to make a mess with it if you don’t fully understand what it does. It is rarely needed and should be rarely used. Code containing `become:` should be viewed with suspicion since there typically is no need to swap the definitions of two objects in real life.

But see [5.17] for a few odd examples where it really is OK.

5.19 • How do I find and remove ’lost’ instances?

Sometimes instances get lost; they are still in the image but nothing you can access proints to them. This can be a frequent occurrence on some implementations when debugging code that opens windows.

Such instances can easily be found with this bit of code:

```
TheClass allInstances
```

Example 45, Evaluate

Unwanted instances can be removed using the technique in Example 46.

```
TheClass allInstances
   do: [ :instance |
      instance become: String new ]
```

Example 46, Evaluate

Some sources recommend using nil instead of a new instance of some class, as in Example 47.

```
TheClass allInstances
   do: [ :instance |
      instance become: nil  "WRONG" ]
```

Example 47, Evaluate

There is a problem with the technique in Example 47 if the code is run on a system with symmetric `become:` because it swaps all instances of `nil` with the unwanted object. Now, `nil` may be a special object encoded in the object pointer†, in which case this may still work, or `nil` may not be a special object, in which case this may cause great rumbling and even flames in the innards of the image. On VisualWorks 2.0, the expression in Example 48 immediately brings the system to a screeching halt.

†. See [12.15] *’What things are encoded in an object pointer?’* on page 174.

```
" I’m bad; do not evaluate me! "
String new become: nil
```
Example 48

Frequently Asked Question

5.20 • How fast is `become:`?

Below are tests of the three implementations I have handy: IBM Smalltalk V3 on a 486 under Warp, VisualWorks 2.0 on a Mac Quadra 950 (68040), and Digitalk Smalltalk/V-Mac on the same Quadra.

I ran the same tests on each, except that the block local variables had to become method local variables in V-Mac. First is the `become:` test:

```
Time millisecondsToRun: [
   1000 timesRepeat: [
      | a b |
      a := 'asdf' copy.
      b := a.
      a become: 'qwer' copy.
      a = b ifFalse: [ self error: 'Whoops!'].
      b    ] ]
```

It creates an object, assigns it to two variables (so to speak), and then does a become with a similar object. When done, it tests to see if the two objects have the same value. If not, something went astray. It answers `b`, which has the same value as `a`. (Select the innermost block contents and evaluate it.)

A similar test uses assignment instead:

```
Time millisecondsToRun: [
   1000 timesRepeat: [
      | a b |
      a := 'asdf' copy.
      b := a.
      a := 'qwer' copy.
      a = b ifTrue: [ self error: 'Whoops!'].
      b    ] ]
```

Since it uses assignment, `b` is unchanged and the test is reversed.

Results

Implementation	Ratio
IBM Smalltalk V3	100:1
VisualWorks 2.0	8:1.
Smalltalk/V-MAC	40:1.

Conclusions

IBM and Digitalk got 100:1 and 40:1 which indicates some big slowness is involved. VisualWorks is only 9:1 which is a lot faster. So, what's going on?

A poster on `comp.lang.smalltalk` indicates that (based on experience implementing `become:`) "it is not much more difficult to do a two-way `become:` than to do a one-way `become:` in the absence of an object table since most of the time is spent searching for objects that may have one of the pointers rather than comparing pointers. VisualWorks does have an object table, but the longer time seems to be due to the object table entries containg all the information that many other systems put in the object header. Interactions with the garbage collector also can add to the time."

"Digitalk and IBM Smalltalk use direct object pointers but may be able to use information from the garbage collector to speed things up. For 'new' objects, the garbage collector holds a pointer to all references to the object from older objects, thus requiring that the full scan only occur in new space."

—Who was this??????????

Frequently Asked Question

5.21 • Is there a simple example of a good use of `become:`?

The subject of caching calculated values comes up now and then. There is a cute way to approach this problem using become:.

A class, `Lazily`, takes a block as a parameter and remembers it.

```
bridgeSize := Lazily do: [ self computeBridgeSize ]
```

This can be done in some initialization phase of an object. When the value is needed, it is sent the value message:

```
bridgeSize value
```

The first time time occurs, the block is evaluated and the result is remembered. All following times the remembered value is answered. The performance is high, since for each of these succeeding times the value message consists of a single return statement; it is no more costly than any other 'getter' method. †

See [10.3] *'How can I defer calculations until they are needed?'* on page 154 for others solutions.

Example

Two lazy values are defined and remembered in `c1` and `c2`:

```
| c1 c2 |
c1 := Lazily do: [2 + 3].           " Define a lazy value "
c2 := Lazily do: [c1 lazyValue * 3]." Define lazy value using c1 "
^ c2 lazyValue * c2 lazyValue       " Evaluate lazy values "
" Answers: 225 "
```

Example 49, Display

Note that in the expression:

```
c2 lazyValue * c2 lazyValue
```

the first evaluation of `c2 lazyValue` must evaluate the block for `c2`, which in turn evaluates the block for `c1`. The second evaluation of `c2 lazyValue` simply returns the integer `15`.

The Magic

There is only one bit of magic; it involves `become:` in one of its good uses.

There are two important classes: `Lazily`, and `LazyEvaluationPerformed`. Class `Lazily` is used as shown above; an instance holds the block but does not evaluate it until asked. An instance of class `LazyEvaluationPerformed` holds the value of the block after it is evaluated.

The implementation of `Lazily>>lazyValue` contains the only interesting code:

†. Of course, depending on your environment, the first time may be quite expensive.

```
lazyValue
    | val cache |
    val := aBlock value.
    cache := LazyEvaluationPerformed new
                  lazyValue: val.
    self become: cache.
    ^ val
```

The block is evaluated and a new instance (`cache`) of `LazyEvaluationPerformed` is obtained with the block's value as its new value. Then cache is swapped with `self` so that all pointers to instances of `self` (of which there is typically one) now point to cache. It does not matter whether the `become:` is symetric or asymetric. In either case, by the end of the `lazyValue` method, the former value of `self` is garbage.

The Code

```
Object
   subclass: #Lazily
   instanceVariableNames: 'aBlock'
   classVariableNames: ''
   poolDictionaries: '' !

Object
   subclass: #LazyEvaluationPerformed
   instanceVariableNames: 'value'
   classVariableNames: ''
   poolDictionaries: '' !

!Lazily class methods!
do: aBlock
   ^ self new
      lazyBlock: aBlock! !

!Lazily methods!
lazyBlock: block
   "PRIVATE"
   aBlock := block!
lazyValue
   | val cache |
   val := aBlock value.
   cache := LazyEvaluationPerformed new
```

```
          lazyValue: val.
    self become: cache.
    ^ val! !

!LazyEvaluationPerformed methods!
lazyValue
    ^ value!
lazyValue: anObject
    value := anObject! !
```

Example 50, Filein

Part VI: Collections

General Questions

Frequently Asked Question

6.1 • What is a collection?

A collection is an object which holds other objects and provides protocol for enumerating through the held objects. Collections are organized in a number of different ways including:

- Indexed or not indexed;
- Indexed by explicit or implicit keys; or
- Holds one kind of object or varied kinds of objects;
- has some ordering to the elements, such as an index, or has no ordering.
- Fixed in size or expandable.

All have in common the basic idea of holding other objects.

Frequently Asked Question

6.2 • What kinds of collections are there?

The exact list of collections is implementation dependent; in fact, many applications provide additional kinds of collections. The collections that are common to most implementations are shown below.

Collection Name	**Description**
`Collection`	Abstract parent of all collection classes
`Bag`	Unordered & unstructured; holds anything but `nil`
`Dictionary`	Holds key/value pairs (Associations); keys are arbitrary
`IdentityDictionary`	Holds key/value pairs (Associations); keys are symbols
`Set`	Like `Bag` but holds just one of any object but `nil`
`IndexedCollection`	Abstract parent of indexed classes; name varies

FixedSizedCollection	Abstract parent of fixed sized classes; name varies
`Array`	Holds anything
`ByteArray`	Holds integers in range 0 to 255.
`String`	Holds characters
`Symbol`	Holds characters; symbols are identity objects
`Interval`	Pseudo-collection representing range of numbers
VariableSizedCollection	Abstract parent of variable sized collections; name varies
`OrderedCollection`	Add anything anywhere; acts like array, queue, stack
`SortedCollection`	Always stays sorted; holds anything

Frequently Asked Question

6.3 • What are Bags and Sets?

Bags and sets are instances of class `Bag` or `Set`. Neither provides any ordering. Both are just big accumulations of stuff. They differ only when adding duplicate objects (as determined by #=). Bags hold as many of one object as are added; sets always hold just one.

In Example 51, a new bag is created and elements from a literal array are added.

```
" Create new Bag and add elements"
| bag |
bag := Bag new.
bag add: #( 1 2 3 4 5 4 3 2 1 ).
^ bag
" Answers: Bag( 1 1 2 2 3 3 4 4 5 ) "
```

Example 51, Evaluate

In Example 52, a new set is created and elements from a literal array are added. The answer shows that the set only holds one of each element.

```
" Create new Set and add elements"
| set |
set := Set new.
set add: #( 1 2 3 4 5 4 3 2 1 ).
^ set
" Answers: Set( 1 2 3 4 5 ) "
```

Example 52, Evaluate

See “Bag and Set Questions” on page 126 for more information.

Frequently Asked Question

6.4 • What are dictionaries?

Dictionaries are collections that hold key/value pairs called associations. (Associations are instances of class `Association`, but it is rarely useful to use associations directly).

Keys in dictionaries can be any object but are usually strings or symbols. Values are added to a dictionary with an associated key and are retrieved by using the key as in Example 53.

```
" Create a dictionary and add elements "
| dict |
dict := Dictionary new.
dict at: 'Fido' put: 'Mutt'.
dict at: 'Spot' put: 'Dalmation'.
dict at: 'Blackie' put: 'Labrador'.
^ dict at: 'Fido'
" Answers: 'Mutt'  "
```

Example 53, Display

Identity dictionaries are dictionaries in which the keys must be identity objects, usually symbols.

See “Dictionary Questions” on page 125 for further information.

Frequently Asked Question

6.5 • What are Indexed Collections?

Indexed collections are collections with integer indexes. Elements are retrieved, and changed using these indexes. Class `IndexedCollection`, which may be named differently in your system, is the abstract parent class of all indexed collections.

Frequently Asked Question

6.6 • What are Arrays?

Arrays are indexed collections that are fixed in size. Arrays are created with a given number of elements, and cannot grow. Each element is initialized to `nil`.

```
| a |
a := Array new: 8.
1 to: a size do: [ :n |
   a at: n put: n ].
^ a
" Answers:  (1 2 3 4 5 6 7 8 nil nil)
```

Example 54, Evaluate

See “Array Questions” on page 126 for more information.

Frequently Asked Question

6.7 • What are OrderedCollections?

Ordered collections are indexed collections that are variable in size. Ordered collections are created with a given number of elements, or a default, and can grow as new elements are added.

```
| oc s1 s2 |
oc := OrderedCollection new: 10.
1 to: 5 do: [ :n |
   oc add: n ].
^ oc
" Answers:   OrderedCollection(1 2 3 4 5 ) "
```

Example 55, Display

See “OrderedCollection Questions” on page 127 for more information.

Frequently Asked Question

6.8 • What are ByteArrays, Strings, and Symbols?

Byte arrays, strings, and symbols are similar in concept. Each is a fixed size, indexed collection of one kind of object. Byte arrays hold integers in the range 0 to 255. Symbols and strings hold characters.

See “String Questions” on page 129 and “Symbol Questions” on page 131 for more information.

Frequently Asked Question

6.9 • What are intervals?

Intervals are objects that act like read-only collections. Their contents are computed as needed.

Intervals are created by specifying a range of numeric values, a start and end value, and an increment. Elements in the interval start with the starting value, with succeeding elements being the previous plus the increment. The ending value is greater-than or equal to the highest value in the interval.

```
| collection oc |
collection := 1 to: 20 by: 2.
oc := OrderedCollection new.
oc addAll: collection.
^ oc
" Answers: OrderedCollection(1 3 5 7 9 11 13 15 17 19) "
```

Example 56, Evaluate

See “OrderedCollection Questions” on page 127 for more information.

Frequently Asked Question

6.10 • How do I pick a collection?

Later◄

General Protocol Questions

Frequently Asked Question

6.11 • What is the basic collection protocol?

All collections support the basic collection protocol, which includes messages for creation, iteration, testing, and conversion.

Creation Messages

aCollectionClass new

Create a new instance of the named class; allocate some default amount of space to hold elements. It is generally useful only for collections that grow.

```
Set new
```

Example 57, Display

```
OrderedCollection new
```

Example 58, Display

aCollectionClass new: size

Create a new instance of the named class; allocate the specified amount of space to hold elements.

```
Array new: 20
```
Example 59, Display

```
OrderedCollection new: 100
```
Example 60, Display

Iteration Messages

In each case, except where noted, pass successive elements of `aCollection` to a one parameter block for some kind of processing.When a collection has an ordering, pass the elements in the order of the collection. A return value is sometimes expected from the block.

`aCollection do: aBlock`

Pass the elements to the block, ignoring return values.

```
#( 1 2 3 8 9 )
   do: [ :element |
      Transcript show: element printString ]
```
Example 61, Evaluate

`aCollection collect: aBlock`

Form a new collection of the same kind as `aCollection` and holding the values answered by the block.

```
#( 1 2 3 8 9 )
   collect: [ :element | element + 1 ]
" Answers: (2 3 4 9 10)
```
Example 62, Display

`aCollection detect: aBlock`

Answer the first element that causes `aBlock` to answer `true`.

```
#( -2 0 2 )
   detect: [ :element | element > 0 ]
" Answers: 2 "
```
Example 63, Display

`aCollection detect: aBlock ifNone: errorBlock`

Answer the first element that causes `aBlock` to answer `true`. If there is no such value, invoke `errorBlock` and answer its result.

```
#( -2 0 2 )
   detect: [ :element | element > 0 ]
   ifNone: [ nil ].
" Answers: 2 "
```
Example 64, Display

`aCollection inject: aValue into: aBlock`

Answer a value formed from an initial value, `aValue`, and the elements of `aCollection`. For each element in the collection, invoke the two-parameter block `aBlock` passing, first, `aValue` and the first collection element, and then for each successive invocation, the prior block result and the next collection element.

```
" Sum the elements in an array "
#( 1 2 3 )
   inject: 0
   into: [ :sum :element | sum + element ].
" Answers: 6 "
```
Example 65, Display

See “inject:into:” on page 119 for more information.

`aCollection reject: aBlock`

Answer a new collection of the same kind as the original holding each element for which `aBlock` answers `false`.

```
#( -2 0 2 )
   reject: [ :element | element >= 0 ].
" Answers: ( -2 ) "
```
Example 66, Display

`aCollection select: aBlock`

Answer a new collection of the same kind as the original holding each element for which `aBlock` answers `true`.

```
#( -2 0 2 )
   select: [ :element | element >= 0 ].
" Answers: ( 0 2 ) "
```
Example 67, Display

Conversion Messages

The conversion messages are: `asArray`, `asBag`, `asByteArray`, `asOrderedCollection`, `asSet`, `asSortedCollection`, and `asSortedCollection:`. Each kind of collection responds to the messages by converting itself to an instance of the appropriate class.

Testing Messages

`aCollection occurrencesOf: anObject`

Answer the number of times `anObject` occurs in `aCollection`.

`aCollection size`

Answer the number of elements in the collection. Note that the current internal size of a variable sized collection may be larger due to space that has been allocated but not yet used.

aCollection isEmpty

Answer `true` if the size is zero.

Frequently Asked Question

6.12 • What is the basic indexed collection protocol?

Frequently Asked Question

6.13 • How can I best add to or remove from collection while I iterate over it?

The best way to add or remove elements from a collection while iterating over it is not to add or remove elements from a collection while iterating over it. It is a bad practice that can get you in deep trouble, sometimes even months later when code that seemed to work suddenly develops very odd symptoms.

Even the so-called unordered collections actually do have some internal ordering, which may not be at all obvious. Even if known, the details might change by vendor or release, or with the size of the collection.

Here are some examples in which each element in a collection is to be added to itself again, directly into the original collection. You may get different results in your implementation since the effects are very implementation dependent.

The first example uses a bag, and appears to work correctly in the implementation used here.

```
" Wrongly duplicating elements in a Bag "
| a |
a := #(1 2 3 4 5) asBag.
a do: [ :elem |
    a add: elem ].
^ a
" Answers: Bag(1 1 2 2 3 3 4 4 5 5 ) "
```

Example 68, Evaluate

But take a look at what happens when the bag becomes an ordered collection:

```
" Wrongly duplicating elements in an OrderedCollection "
| a |
a := #(1 2 3 4 5) asOrderedCollection.
a do: [ :elem |
    a add: elem ].
^ a
" Answers:
  OrderedCollection(1 2 3 4 5 1 nil nil nil 1) "
```
Example 69, Display

Even more bizare results are found when something similar is coded for a dictionary in Example 70. First a dictionary is created with a key of `'1'` having a value of `1`, a key of `'2'` having a value of `2`, and so on. Then an attempt is made to add another key for each key in the dictionary; the new key is the old key with an `'x'` appended and the new value is the old value multipled by `10`. Finally, the results are written to the `Transcript`.

```
" WRONG: Add elements to a dictionary while iterating thru it "
| d |
d := Dictionary new.
#(1 2 3 4 5) do: [ :elem |
    d at: elem printString put: elem ].
d keysDo: [ :key |
    d at: key,'x' put: (d at: key)*10 ].
d keysDo: [ :key |
    Transcript show: key, ':', (d at: key) printString, ' ' ]
" Transcript contains:
1x:10 2x:20 3x:30 4x:40 5x:50 1xx:100 1:1 2:2 3:3 4:4 5:5
2xx:200 3xx:300 4xx:400 "
```
Example 70, Display

There are fourteen elements in the dictionary, not ten as expected; some added elements were themselves processed. How can this be?

Some kinds of collections grow or shrink when elements are added or removed, potentially causing elements to be moved around. This may cause elements to be skipped, or to be processed more than once. This is what happened with the ordered collection example.

Dictionaries are hashed and new elements go wherever they happen to go. When iterating through the dictionary some internal representation of the keys is used which does have some ordering, meaningful only to other parts of the innards, but potentially causing bugs when updating at the same time.

The solution is simple: never update a collection that you are iterating over. Always make a copy of the collection first, then iterate over that. Here are the `Bag` and `OrderedCollection` examples coded the right way:

```
" Properly duplicating elements in a Bag "
| a |
a := #(1 2 3 4 5) asBag.
a copy do: [ :elem |
    a add: elem ].
^ a
" Answers: Bag(1 1 2 2 3 3 4 4 5 5 ) "
```

Example 71, Evaluate

```
" Properly duplicating elements in an OrderedCollection "
| a |
a := #(1 2 3 4 5) asOrderedCollection.
a copy do: [ :elem |
    a add: elem ].
^ a
" Answers:
   OrderedCollection(1 2 3 4 5 1 2 3 4 5 ) "
```

Example 72, Display

Dictionaries can be done the same way, but there is a neat trick. Instead of copying the whole dictionary, ask for a collection holding its keys. Then iterate over that collection.

```
" Properly duplicating elements in a Dictionary "
| d |
d := Dictionary new.
#(1 2 3 4 5) do: [ :elem |
    d at: elem printString put: elem ].
d keys do: [ :key |
    d at: key,'x' put: (d at: key)*10 ].
d keysDo: [ :key |
    Transcript show: key, ':', (d at: key) printString, ' ' ]
" Transcript contains:
   5x:50 1:1 2:2 3:3 4:4 5:5 1x:10 2x:20 3x:30 4x:40 "
```

Example 73, Display

—Adapted from and inspired by Ralph Johnson's, Classic Smalltalk Bugs

Using the add: Message

Frequently Asked Question

6.14 • How do I use the add: message?

On the surface, Example 74 should answer an instance of `Set` with two elements; instead, it gets a walkback complaining that the integer `1` does not understand `add:`.

```
" Being burnt by add: "
| items |
items := Set new
   add: 1.
items add: 2.
^ items
```

Example 74, Display

The problem here is that the `add:` message answers its parameter, rather than `self`. Thus, the variable `items` is set to an integer `1` in the third line. In the fourth line, the `add:` message is sent to the value in items, the integer `1`.

This is one of the places to use `yourself`:

```
" Using the yourself message "
| items |
items := Set new
   add: 1;
   yourself.
items add: 2.
^ items
```

Example 75, Display

or, better:

```
" Using the yourself message "
^ Set new
   add: 1;
   add: 2;
   yourself
```

Example 76, Display

6.15 • Are there any good reasons why add: returns its parameter?

The answer generally given is that answering the added object allows that object to be further manipulated without having to assign it to a variable.

However, I, for one, do not recall ever having done that in 13 years of coding Smalltalk, and have been burnt a number of times when I forget how it works. I certainly have coded a lot of `yourself` messages to overcome the 'feature'.

I recently skimmed quickly through all sends of `add:` in the Digitalk V/Mac 2.0 image, and I looked at about 100 sends in VisualWorks 2.0. I found no instances where code in the image did anything with the answered value. I found many cases in the Digitalk image where `yourself` was sent to overcome this ’feature’.

What could the reason be? Is there something useful that could be being done? Lets look at some examples.

In Example 77, the string is added to the collection, and answered. The string is then initialized with asterisks. This is very unclear; even an very experienced Smalltalk programmer may have to look twice to be sure what it is doing.

```
" Add, then initialize "
| coll |
coll := OrderedCollection new.
(coll add: (String new: 20))
   atAllPut: $*
```

Example 77, Evaluate

In Example 78, the string is initialized, then added to the collection. This is cleaner and more logical than in Example 77 and doesn’t need `add:` to return its parameter.

```
" Initialize, then add "
| coll |
coll := OrderedCollection new.
coll add: ((String new: 20) atAllPut: $*)
```

Example 78, Evaluate

In Example 79, the string is added to the collection, answered, and assigned. The string is then initialized with asterisks.

```
" Add, save, then initialize "
| coll str |
coll := OrderedCollection new.
str := coll add: (String new: 100).
str atAllPut: $*
```
Example 79, Evaluate

In Example 80, the string is initialized, then added to the collection. This is cleaner and more logical than in Example 79 and doesn't need `add:` to return its parameter.

```
| coll str |
coll := OrderedCollection new.
str := (String new: 100) atAllPut: $*.
coll add: str
```
Example 80, Evaluate

None of the alternatives above that use the 'feature' are as good as those that don't.

inject:into:

The `inject:into:` messages to collections are rarely used but do not deserve their obscurity. The questions below illustrate some of the many ways that `inject:into:` can be used.

Frequently Asked Question

6.16 • How is inject:into: used for summing?

Given a collection of numbers, sum the numbers.

```
" Simple inject:into: to sum elements in an array "
| array sum |
array := #( 1 2 3 4 5 ).
sum := array
    inject: 0
    into: [ :inj :ele |
        inj + ele ]
" Answers: 15 "
```
Example 81, Display

What happens here? Figure 4 shows each step of the iteration. The result of each step becomes the injected value for the next step.

Iteration	injected value	Element value	Result
1	0	1	1
2	1	2	3
3	3	3	6
4	6	4	10
5	10	5	15

Figure 4: Iterations for inject:into: in Example 82.

The summing can be of a property of the collection. In Example 82 the collection holds strings and the expressions answers the total length of all of the strings.

```
" Compute the total length of a collection of strings "
| arrayOfStrings |
arrayOfStrings := #('asdf' 'qwer' 'wert' '1234' ).
arrayOfStrings
      inject: 0
      into: [ :len :str | len + str size].
" Answers: 16 "
```

Example 82, Display

Frequently Asked Question

6.17 • How is inject:into: used to form products?

Products can be formed with `inject:into:` as well as sums. A common example computation is the factorial of an integer. Example 83 shows how to comput factorial with `inject:into:`.While this may be slower than the builtin `factorial` method, the same technique can be used on other series for which no method is provided.

```
" n Factorial using inject:into: "
| n |
n := 10.
(1 to: n)
    inject: 1
    into: [ :inj :ele |
        inj * ele ]
" Answer: 3628800 "
```

Example 83, Display

6.18 • Can collections be injected?

Anything can be injected. Sometimes it is useful to inject a collection. One such case involves the calculation of the Fibonacci series.

The Fibonacci series starts with the two values 1 and 1; each successive value is the sum of the previous two elements:

```
1 1 2 3 5 8 13 21 34 55 89 ...
```

The Fibonacci series requires memory of the previous two elements to compute the current element, but `inject:into:` appears to provide but one other value, that which is injected. However, since the injected value can be *anything*, including a collection, we'll inject the collection that holds the series itself and simply add each new element to the end.

```
" Fibonacci using inject:into: "
| nterms |
nterms := 10.
^ (3 to: nterms)
    inject: (OrderedCollection with: 1 with: 1)
    into: [ :inj :n |
        inj addLast: (
            (inj at: n - 2) +
            (inj at: n - 1) ); yourself ].
" Result: 1 1 2 3 5 8 13 21 34 55 "
```

Example 84, Display

Another example where injecting a collection is useful involves accumulating a collection of words into a string, separating words with a blank. An empty string (which is, after all, a collection) is injected.

```
" Given an array of words, make a sentence fragment "
| words |
words := #( 'Gone' 'With' 'the' 'Wind' ).
words
    inject: ''
    into: [ :inj :ele |
        inj, ele, ' ' ]
" Answers: 'Gone with the wind '  "
```

Example 85, Display

6.19 • Can inject:into: simulate collect: and select:?

Several standard collection iterators can be simulated with `inject:into:`. Simulations for the `collect:` and `select:` iterators are shown below. In each case the normal usage is shown followed by an `inject:into:` equivalent.

Collect

```
" collect: "
stuff collect: [ :element | element + 1]
```

Example 86

```
"collect: simulated with inject:into: "
| stuff |
stuff := #( 1 2 3 4 5 ) asOrderedCollection.
stuff inject: (stuff class new: stuff size)
      into: [ :result :element |
          result add: element + 1; yourself ]
" Answers: OrderedCollection(2 3 4 5 6 ) "
```

Example 87, Display

Select

```
" Select: "
stuff select: [ :element | element > 0 ]
```

Example 88

```
" select: simulated with inject:into: "
| stuff |
stuff := #( 1 2 3 4 5 ) asOrderedCollection.
stuff inject: stuff class new
      into: [ :result :element |
          element > 0
              ifTrue: [ result add: element; yourself ] ]
 " Answers: OrderedCollection(1 2 3 4 5 ) "
```

Example 89, Display

Note that `collect:`, and `select:` have defaults for the class of the collection they return, and that the `inject:into:` versions simulate this by making the result the same as that of the original collection. (This is not always true though; for example, the simulation won't work if `stuff` is an Array).

One advantage to using `inject:into:` is the specification of the resulting collection. Here is the `collect:` simulation again in which the resulting collection is specified.

```
| stuff |
stuff := #( 5 1 4 2 3 ) asOrderedCollection.
stuff inject: (SortedCollection new: stuff size)
      into: [ :result :element |
          result add: element + 1; yourself
" Answers:  SortedCollection(2 3 4 5 6 ) "
```

Example 90, Display

Frequently Asked Question

6.20 • How can inject:into: be used to find a maximum value?

Instead of summing the values in or other characteristics of a collection, the injected value might be some other mathematical operation such as the maximum value. Example 91 finds the maximum value in a collection of positive numbers. The critical code is in bold.

```
" Find the largest value in a collection using inject:into: "
| c |
c := #( 1 4 3 6 9 8 2 ).
^ c inject: 0
    into: [ :max :elem |
        max max: elem ]
" Answers: 9 "
```

Example 91, Display

Example 92 shows an alternative using `do:` which is similar in length but requries the addition of a local variable.

```
" Find the largest value in a collection using do: "
| c max |
c := #( 1 4 3 6 9 8 2 ).
max := 0.
c do: [ :elem |
    max := max max: elem ].
^ max
" Answers: 9 "
```

Example 92, Display

Hashed Collection Questions

Frequently Asked Question

6.21 • Why won’t my instances work right in Sets, Bag, or Dictionaries?

Sets, bags, and dictionaries are all hashed collections. User defined instances must implement equal (=) and `hash` methods so that the values work properly in hashed collections.

See [6.22] *’How do hashed collections work?’* for details.

Frequently Asked Question

6.22 • How do hashed collections work?

Collections that use hashing to find elements include sets, bags, and dictionaries. These collections use the value answered by the `hash` method as a part of an indexing operation to find elements.

Hash values are integers computed from the data of the object. Integers, for example, usually just answer themselves and strings answer a value computed from the internal value of the characters. Hash values are used for two things:

- A quick comparison for (possible) equality, and
- The source of an value used as the index into a collection.

For example, a string stored in a bag is actually stored in an indexed collection, such as an array, at a position that is calculated similar to this:

```
position := (aString hash \\ bagIndexedCollection size) + 1
```

Then, the value, `position`, is used as an index into `bagIndexedCollection` for the place where the value is to be put. Since hash values are usually not unique, it is necessary to compare, using equal, any object already in the collection with the new value. If the comparison is true, then the object is already in the collection; if false, it is not.†

†. When the values are not equal, it called a collision; collision management is a topic beyond the scope of this question.

Thus, if equal values do not produce equal hashes, hashed collections will not work. Objects in Smalltalk must thus always obey the rule:

Equal objects must have equal hash values.

Frequently Asked Question

6.23 • How do I reimplement a `hash` method?

If an object reimplements the = method it *must* reimplement the `hash` method.

For example, consider a new object that holds a US-style phone number. It has three components: area code, exchange, and number. The equal method is:

```
= aPhoneNumber
   ^ (self areaCode = aPhoneNumber areaCode   and:
     [self exchange = aPhoneNumber exchange]) and:
     [self number  = aPhoneNumber number]
```

Example 93

A new `hash` method must answer a hash that uses all of the components compared by equal. For example, either of the following might be used:

```
hash
   ^ areaCode hash + exchange hash + number hash
```

or:

```
hash
   ^ (areaCode hash xor: exchange hash) xor: number hash
```

Dictionary Questions

Bag and Set Questions

Frequently Asked Question

6.24 • Why isn’t the comma message (#,) implemented for Bags and Sets?

The comma message is implemented for collections that have an ordering. It forms a new collection which has the two component collections placed 'end-to-end'. That is not a meaninful operation with sets since sets or bags have no ordering.

Of course, that doesn't stop you from adding such a message, but it is better to use `addAll:` which does work for sets and bags and for most other collections too. But, note that `addAll:` changes the collection and comma answers a new collection. A comma-equivalent is:

```
(self copy)
   addAll: aCollection;
   yourself
```

Example 94

Indexed Collection Questions

Array Questions

OrderedCollection Questions

Frequently Asked Question

6.25 • What is the size of an OrderedCollection?

The size of an ordered collection is the number of elements actually in the collection, not the potential size of the collection, which might be larger. Example 95 shows this by adding the size to the collection twice, once before any other elements are added and once after adding all other elements. The sizes are shown in bold.

```
" Examples of size of an OrderedCollection "
| oc s1 s2 |
oc := OrderedCollection new: 10.
oc add: oc size.
1 to: 5 do: [ :n |
   oc add: n*10 ].
oc add: oc size.
^ oc
" Answers: OrderedCollection(0 10 20 30 40 50 6 ) "
```

Example 95, Display

Frequently Asked Question

6.26 • How can the capacity of an OrderedCollection be determined?

The capacity of an ordered collection is the number of elements that can be inserted before it needs to grow. This is the value provided with `new:` when a new ordered collection is created.

Some implementations provide a `capacity` message which answers the number of potential elements the collection can hold:

```
oc capacity
```

In IBM Smalltalk, the data in an ordered collection is held within another collection. The `elements` message answers this other collection; its size is the capacity of the ordered collection:

```
oc elements size
```

SortedCollection Questions

Frequently Asked Question

6.27 • Is it better to add to a SortedCollection or sort a built collection?

Which is better, if either:

```
inProperOrder: aCollection
   | sc |
   sc := SortedCollection new.
   sc addAll: aCollection
```

or:

```
inProperOrder: aCollection
   | sc |
   sc := aCollection asSortedCollection
```

Here are two tests:

```
" Add to sorted collection "
| sc aCollection |
aCollection := (1 to: 500 by: 3),
   (1 to: 500 by: 5), (1 to: 500 by: 7).
Time millisecondsToRun: [
   10 timesRepeat: [
     sc := SortedCollection new.
      sc addAll: aCollection ]   ]
" Results:
      VW 2.0:     636 612   (Mac)
      Digitalk: 1117  1100   (V/Mac 2.0) "
```

Example 96, Display

```
" Convert to sorted collection "
| sc aCollection |
aCollection := (1 to: 500 by: 3),
   (1 to: 500 by: 5), (1 to: 500 by: 7).
Time millisecondsToRun: [
   10 timesRepeat: [
     sc := aCollection asSortedCollection  ]   ]
" Results:
      VW 2.0:    468 472    (Mac)
      Digitalk: 1100 1117   (V/Mac 2.0) "
```

Example 97, Display

The timings are similar. On the particular implementations shown, it is faster to convert using VW than to `add:` but on Digitalk the timings are essentially the same.

String Questions

Frequently Asked Question

6.28 • When is it better to use streams or concatenation to build a new string?

Let’s try two cases, one with concatenation and one with streams.

```
" Build a string holding 100 ’x’ characters using concatenation "
Time millisecondsToRun: [
   10000 timesRepeat: [
      | s |
      s := 'x'.
      99 timesRepeat: [ s := s , 'x'  ] ]  ]
" Results:  47629 45480 "
```

Example 98, Display

```
" Build a string holding 100 ’x’ characters using a stream "
Time millisecondsToRun: [
   10000 timesRepeat: [
      | s |
      s := WriteStream on: (String new: 100).
      100 timesRepeat: [ s nextPutAll: 'x' ]
" Results:  23281 23085 "
```

Example 99, Display

The version using concatenation takes twice the time. Why?? Each concatenation causes a new string to be allocated. It is the length of the two parts. The old parts may become garbage. This is repeated 99 times:

```
1:  x,x         ==> xx
2:  xx,x        ==> xxx
3:  xxx,x       ==> xxxx
4:  xxxx,x      ==> xxxxx
5:  xxxxx,x     ==> xxxxxx
6:  xxxxxx,x    ==> xxxxxxx
7:  xxxxxxx,x   ==> xxxxxxxx
8:  xxxxxxxx,x  ==> xxxxxxxxx
```

However, in the stream, the values are appended to the string that is the stream contents. (If that string is too short, it will be expanded and the old one thrown away, but that won't happen very many times even if you start with a stream on a string of length 1.)

Both methods have to copy the new string to the end of something. Concatenation also has to copy the old string to the front of a slightly larger new string. As it gets bigger and bigger, that becomes an overwhelming cost.

Basically, a stream continues to add on new stuff behind previous stuff in a large working collection (called the stream contents). If the working collection fills up, the stream will then allocate a bigger collection and do a copy of the data to it.

Frequently Asked Question

6.29 • How do I convert numbers to strings and strings to numbers?

Smalltalk implementations have limited provision for conversion of numbers to string and strings to numbers.

The `asNumber` Method

Many implementations support the `asNumber` method of class `Number` but the details if implementation may differ. Typically, the method converts any leading number characters to an integer and answers the integer, as in Example 100.

```
" Convert a string to a number "
'1234' asNumber
" Answers: 1234 "
```

Example 100, Display

Some will ignore non-numeric characters, as in Example 101.

```
" Convert a string with garbage to a number "
'1234asdf' asNumber
" Answers: 1234 "
```

Example 101, Display

Other Methods

Various implementations provide various conversion methods:

IBM Smalltalk

Digitalk

VisualWorks

Other Code

Code is available from other sources.

IBM Smalltalk

There is extensive support for IBM Smalltalk given in *IBM Smalltalk: The Language*, pages 257-274.

See *IBM Smalltalk: The Language* in question [1.11] on page 6.

Frequently Asked Question

6.30 • Can I do regular expression matching in Smalltalk?

There is no builtin facility for regular expressions in Smalltalk.

Symbol Questions

Frequently Asked Question

6.31 • What is a symbol?

Frequently Asked Question

6.32 • Why are there symbols?

6.33 • Is comparing symbols faster than comparing strings?

Yes. Symbols require only a comparison of object pointers, and that is very fast. Comparison of string requires a comparison of characters in the strings.

It is often even faster to use == to compare symbols since == is often implemented directly by the compiler.

However, making symbols is an expensive operation. One should not write code like this:

```
aString asSymbol == #Cancel ifTrue: [ ... ]
```

While the comparison is fast the conversion of `aString` to a symbol is slow.

Symbols are identity objects. It is guaranteed that two symbols that have equal values are always the same object. Thus, if two symbols compare equal with = then they will always compare equal with ==. Since symbols can be built on the fly, and can contain any characters, some amount of magic is required to make them work.

This magic varies from implementation to implementation, but basically depends on a table which holds the symbols. The hash of a symbol, which is guaranteed to be unique among all symbols, is the index into the table. The object pointer to a symbol is to the table entry.

To illlustrate the timing differences, the definitions in Example 102 are compared as shown in Example 102. †

```
symbolOne := symbolTwo := #TestSymbol.
stringOne := stringTwo := 'TestSymbol'
```

Example 102

Expression	V-Mac	VW2.0	VSE	IBM 3.0
`symbolOne == symbolTwo`	1	1		
`stringOne = stringTwo`	9	5		
`stringOne = symbolTwo asString`	9	30		
`symbolOne == stringOne asSymbol`	120	144		
`stringOne asSymbol == stringTwo asSymbol`	243	270		

†. Thanks to William Hollings, hollings@inforamp.net, for the timings for VisualWorks.

Comparing strings is a factor of 5 to 10 slower than comparing symbols, but converting a string to a symbol is more than a hundred times more expensive.

Frequently Asked Question

6.34 • When should I not compare symbols?

Frequently Asked Question

6.35 • Are there alternatives to symbols?

Frequently Asked Question

6.36 • Should I use symbols as dictionary keys?

Frequently Asked Question

6.37 • Are there any symbol pitfalls?

Dictionary Questions

Frequently Asked Question

6.38 • What are pool dictionaries?

Frequently Asked Question

6.39 • What is Smalltalk (the dictionary)?

Frequently Asked Question

6.40 • What kinds of collections can be streamed?

Interval Questions

Frequently Asked Question

6.41 • What are intervals?

```
zot copyReplaceAll: (2 to: 4), (20 to: 22) with: #( 0 0 0 1 1 1)◄
(2 to: 100) select: [ :n | n even ]◄
```

Adding new Collections

Frequently Asked Question

6.42 • How can new kinds of collections be implemented?

Part VII: Magnitudes

Magnitudes

Numbers

Frequently Asked Question

7.1 • What is generality?

Frequently Asked Question

7.2 • What is coersion?

Frequently Asked Question

7.3 • How do I convert numbers to strings?

Most Smalltalk systems do not have good ways to convert numbers, other than integers, to strings. Conversion of floating point numbers is usually terrible, with no way to control the precision or form of the result. Fractions can be printed only as the ratio of two numbers without converting to a float first; either choice is bad since the first is usually unintelligable to humans and the second looses precision.

The Messages

`printString`

This message converts anything to some kind of printable representation. Here are some sample conversions:

```
1.2 printString
" Result: ’1.2’  "
```

Example 103, Display

```
1.2345678901234567890 printString
" Result: ’1.2345678’ "
```

Example 104, Display

*(*depends on vendor; show all*)*◄

```
(2/3) printString
" Result: ’(2/3)’ "
```

Example 105, Display

The fraction:

```
    1572584048032818633353217 / 1111984844349868137938112
```

is actually an approximation of the square root of two to ’lots’ of digits but it isn’t obvious when it is printed as a fraction. When printed in decimal it is more clearly the square root of two:

```
    xxxxxxx
```

(What else and on which platforms?)◄

Integers

Frequently Asked Question

7.4 • How big are SmallIntegers?

Frequently Asked Question

7.5 • Can Smalltalk really do bit twiddling?

Frequently Asked Question

7.6 • How can the limitations of a register be simulated?

Floating Point

Frequently Asked Question

7.7 • What information is avavailable about floating-point numbers?

There are a few standard references that every programmer doing serious work with floating-point numbers should have. These include:

- [Goldberg1991] 'What Every Computer Scientist Should Know About Floating-Point Arithmetic', *ACM Computing Surveys*, Vol. 23, Number 1, March 1991, pp 5-48
- [IEEE1985] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985.
- [IEEE1987] *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Standard 854-1987.

The following references describe floating-point arithmetic for Smalltalk.

- [SMITH95] *IBM Smalltalk: The Language*, David N. Smith, Benjamin/Cummings, 1995. ISBN: 0-8053-0908-X. See pages 33, and 449-456.
- (another?)

A num analysis text too◄

Frequently Asked Question

7.8 • Are Floats approximations left over from days of expensive hardware?

Nope, no hold-over. Floats are *exactly the right thing* for scientific and engineering applications. They ain't broke and don't need fixing. People spend millions buying special hardware that does nothing but bang floating-point numbers very rapidly. (Cray's are just giant and expensive floating-point number bangers.) Such systems are measured (informally) in GigaFLOPS, the number of floating point operations per nanosecond. Most scientific or engineering computing would be impossible using 'toy' arithmetic like integers or scaled decimal.

Floating point numbers can be 4 bytes, 8 bytes or longer. ParcPlace VisualWorks uses 4 byte floats as the default, but allows 8 byte floats. Most other Smalltalk systems support only 8-byte floats, though the forthcomming ANSI Standard Smalltalk appears to support 4-, 8-, and 16-byte widths. Many hardware systems, especially if intended for high-speed scientific use, support 'extended' precision, usually 16-byte floats.

Frequently Asked Question

7.9 • What is a floating point number?

Frequently Asked Question

7.10 • What is an IEEE floating point number?

Frequently Asked Question

7.11 • How can I determine numeric precision in my code?

Frequently Asked Question

7.12 • Why don’t Floats give the ’Right Answer’?

Questions about floating point arithmetic frequently look like these:

> "I evaluate `450.61 = (1502.03 - 1051.42)` and the answer is `false` even though `1502.03 - 1051.42` is `450.61`. Why does Smalltalk give the wrong answer?"
>
> "I evaluate `12.000007` truncated and I get `12.000008765476`…. How can <my vendor> ship broken arithmetic?"

Floating point numbers are supposed to be that way. They are designed for use in the engineering and scientific communities where calculations involve wide ranges of number sizes. They are not designed for use in monetary calculations.

It is impossible, in general, to precisely convert decimal numbers to binary, and floats use a binary representation. Thus most numbers are 'wrong' as soon as they are converted.

To see what is actually going on I used some code from [SMITH95] pages 257-274, that prints floating point numbers to any length. I also coded a method which prints IEEE double-precision floats in hex. It produces this format:

```
1.XXXXXXXXXXXXXeEE
```

where ’`x`’ indicates a hex digit in the fraction, ’`e`’ is the literal character ’`e`’, and ’`EE`’ is an exponent of one or two hex digits giving the number of *bits* to shift the fraction. The digit to the left of the decimal point is always a 1, except when the whole value is zero.

Lets look at some examples. First, just some plain numbers:

```
Expression         Result
0.0  printHex      '0e00'
0.3  printHex      '1.3333333333334e-2'
1.0  printHex      '1.0e0'
10.0 printHex      '1.4000000000000e3'
16.0 printHex      '1.0e4'
```
Example 106, Display

Now, the numbers from the question, and one number that's 0.01 larger. Carets mark interesting places:

```
Expression          Result
1502.03 printHex    '1.7781EB851EB85eA
1502.04 printHex    '1.77828F5C28F5CeA'
                          ^
```
Example 107, Display

```
1051.42 printHex    '1.6DAE147AE148eA'
```
Example 108, Display

The `printStringWidth:` method prints numbers to the given character width. The width used here is 20, which means 18 digits (less period and sign); double-precision floating-point provides about 17 decimal digits. As a result, it may show junk past the end of the number. In the cases below, it shows extra zeros.

```
Expression                      Result
1502.03 printStringWidth: 20    '1502.0299999999999970'
1051.42 printStringWidth: 20    '1051.4200000000000070'
```
Example 109, Display

Comparison for equality

A basic rule is:

Never compare floating-point numbers for equality.

Looking at the calculation in Question [7.12], `(1502.03 - 1051.42)`, and calculating the difference, we see that the expected answer `450.61` is not quite what we get:

```
Expression                                 Result
(1502.03 - 1051.42) printStringWidth: 20  '450.60999999999999000'
450.61 printStringWidth: 20               '450.61000000000000100'
                                                ^            ^
```

Example 110, Display

In hex:

```
Expression                                 Result
(1502.03 - 1051.42) printHex              '1.C29C28F5C28F4e8'
450.61 printHex                           '1.C29C28F5C28F6e8'
                                                        ^
```

Example 111, Display

In fact, the difference is just in the rightmost two bits, but that's enough to make it not equal. Now, what do `roundTo:` and `truncateTo:` do? Can we use them to make things come out equal?

```
Expression                                 Result
((1502.03 - 1051.42) roundTo: 0.01)
printStringWidth: 20                      '450.61000000000000100'
                                                ^            ^
((1502.03 - 1051.42) truncateTo: 0.01)
printStringWidth: 20                      '450.60000000000000200'
                                                ^            ^
```

Example 112, Display

Note the digits just above the right-most carets. Clearly, equality testing is not going to work.

Frequently Asked Question

7.13 • How should floating-point numbers be compared?

Since it is not proper to make a comparison such as:

```
 (1502.03-1051.42) = 450.61
```

Example 113, Display

How are floating point numbers best compared? Consider the following:

```
   (1502.03-1051.42) between: 450.605 and: 450.615
```
Example 114, Display

which *does* answer true and *will* answer `true` unless and until the real, internal precision gets very much smaller (or the values get much, much larger). A method which does the `between:and:` test for floats is:

```
!Float publicMethods !
equals: aFloat to: delta
   " Compare self (a float) with aFloat to a 'precision' of delta.
     Example: 1502.03 equals: 1051.42 to: 0.01 "
   | deltah |
   deltah := delta / 2.
   ^ self between: aFloat-deltah
              and: aFloat+deltah  ! !
```
Example 115, Filein

Examples:

```
Expression                                         Result
(1502.03 - 1051.42) equals: 450.61 to: 0.01        true
(1502.03 - 1051.42) equals: 450.61 to: 0.001       true
(1502.03 - 1051.42) equals: 450.61 to: 0.0000001   true
(1502.03 - 1051.42) equals: 450.61 to: 100.0       true

(1502.03 - 1051.42) equals: 450.62 to: 0.01        false
(1502.03 - 1051.42) equals: 450.60 to: 0.01        false
```
Example 116, Display

Frequently Asked Question

7.14 • How is subtraction bad and how can the problem be avoided?

Certain operations can cause the loss of precision. One famous gotcha involves subtracting two floats that are nearly equal. The precision of the result may be only a digit or two or even none at all.

(more)◄

Fractions

Fixed Point

Frequently Asked Question

7.15 • How do I do fixed point calculations in Smalltalk?

Adding New Magnitudes

Frequently Asked Question

7.16 • How about complex numbers?

Frequently Asked Question

7.17 • Where does Vector go?

Frequently Asked Question

7.18 • Where does Money go?

Frequently Asked Question

7.19 • How do I add a subclass to Integer that restricts the range of values?

Here is a very quick and dirty example of how you might approach the problem. The code was tested on Smalltalk/V-Mac 2.0. A real implementation would have to override a number of additional methods inherited from `Integer`.

```
Integer
   subclass: #FixedRangeInteger
   instanceVariableNames: 'value low high'
   classVariableNames: ''
   poolDictionaries: '' !

!FixedRangeInteger class methods!
value: value low: low high: high
   ^ self basicNew
```

```
        value: value
        low: low
        high: high! !

!**FixedRangeInteger methods**!
**+ anInteger**
   "Answer the result of adding the receiver and anInteger"
   | result |
   result := anInteger + value.
   [ result > high ]
       whileTrue: [ result := result - (high - low + 1) ].
   ^ self class value: result low: low high: high!
**- anInteger**
   "Answer the result of subtractiong anInteger from the receiver"
   | result |
   result := value - anInteger.
   [ result < low ]
       whileTrue: [ result := result + (high - low + 1) ].
   ^ self class value: result low: low high: high!
**printOn: aStream**
   "Print the receiver on aStream"
   aStream nextPutAll: value printString, ':';
      nextPutAll: low printString, ':';
      nextPutAll: high printString!
**value: newValue low: lo high: hi**
   "PRIVATE: Reset the values in an instance"
   value := newValue.
   low := lo.
   high := hi! !
```

Example 117, Filein

Example:

```
" Example of fixed range integers "
| r1 r2 r3 |
r1 := FixedRangeInteger value: 6 low: 3 high: 7.
r2 := r1 + 1.
r3 := r2 + 1.
Array with: r1 with: r2 with: r3
" Answers: (6:3:7 7:3:7 3:3:7) "
```

Example 118, Evaluate

Dates and Times

Frequently Asked Question

7.20 • What is a Date?

Frequently Asked Question

7.21 • What is a Time?

Characters

Random Numbers

Frequently Asked Question

7.22 • What is a random number?

Frequently Asked Question

7.23 • What is a random number generator (RNG)?

Frequently Asked Question

7.24 • Does Smalltalk have a random number generator? (where is it?)

Most versions of Smalltalk have random number generators.

(See *7.26 'Is there a good random number generator I can use in Smalltalk?'* for information on and code for the Park-Miller minimum-standard RNG.)

IBM Smalltalk, Release 2 and later

See class `EsRandom`.

Digitalk

Digitalk implementations do not have a random number generator.

ParcPlace VisualWorks 2.0

VW 2.0 claims to have a random number generator, but it is completely broken. Random number generators should not repeat, or start the sequence of numbers over, until a very large sequence of values has been generated. Typically, the sequence should have at least (2^31)-1 distinct random numbers. VW 2.0 repeats after a mere 120000 distinct numbers. The following code illustrates this problem.

```
| randy n |
randy := Random new.
n := 0.
[ randy next ~= 0.0 ] whileTrue: [ ].
[ randy next ~= 0.0 ] whileTrue: [ n := n + 1 ].
Transcript cr; show: 'Period is ', n printString
```

Example 119, Evaluate

Result in Transcript for each run:

```
Period is 120049
```

Frequently Asked Question

7.25 • Are all random number generators the same?

No! There are several critical things to keep in mind:

- Most pseudo-random number generators are terrible.
- Pseudo-random number generators do not really produce random numbers, but produce sequences of very non-random, repeatable, deterministic values. The 'good ones' let their users pretend that, for certain applications, the deterministic sequences are random.
- It is of critical importance to know what you want to do with the pseudo-random

numbers.

One might use a simple, quick and dirty generator to introduce some 'unpredictability' in an action game where performance is critical. But that same generator would be worse than useless with a card game where the user could readily see patterns in the deal after playing a number of games.

A pseudo-random number generator that works well for card games may not work well with a monte-carlo analysis of a large business, and a generator that works for such analysis probably would be horrible for cryptographic use. Good generators for cryptographic use are orders of magnitude more complex and much slower.

Frequently Asked Question

7.26 • Is there a good random number generator I can use in Smalltalk?

There is a paper that proposes a minimal standard random generator. It is 'Random Number Generators: Good Ones Are Hard to Find', by Stephen K. Park and Keith W. Miller in *Communications of the ACM*, 31(10):1192-1201, 1988.

Park and Miller don't claim that theirs is good, but that any generator that is worse should never be used. The meaning of 'good' depends too much on what the generator is to be used for. One can give no opinions about the validity of a generator without testing.† The Park-Miller generator has been extensively tested.

The `ParkMillerRNG` class, described below, is used like this:

```
| randy |
    randy := ParkMillerRNG new.
    10 timesRepeat: [
      Transcript cr;
         show: randy next printString ].
```

Example 120, Display

Creation Methods Summary

`ParkMillerRNG new`	Create a new Park-Miller RNG

†. The testing of random number generators is described in Knuth Vol 2.

Generation Methods Summary

`randy next`	Floating point value: `[0.0,1.0)`
`randy nextInteger`	Integer value: `[0,maxInt]`

The Code

This code was tested and exported from Smaltalk/V-Mac. Essentially similar versions have been tested on IBM Smalltalk.

```
Object
   subclass: #ParkMillerRNG
   instanceVariableNames: 'seed'
   classVariableNames: 'PMa PMm PMmu1 PMq PMr'
   poolDictionaries: '' !

!ParkMillerRNG class methods!
initialize
   PMa   := 16r000041A7.      " magic constant      =      16807 "
   PMm   := 16r7FFFFFFF.      " magic constant      = 2147483647 "
   PMq   := 16r0001F31D.      " quotient (m quo: a) =      44488 "
   PMr   := 16r00000B14.      " remainder (m \\ a). =       2836 "
   PMmu1 := 0.465661287524579 69e-9. " PMmu1 == 'PMm under 1' "!
new
   self initialize.
   ^ super new initialize! !

!ParkMillerRNG methods!
initialize
   " Set a reasonable Park-Miller starting seed "
   seed := 2345678901!
next
   " This method generates random instances of Float
     in the interval 0.0 to 1.0 "
   seed := self peekInteger.
   ^ seed * PMmu1!
nextInteger
   " This method generates random instances of Integer
     in the interval 0 to 16r7FFFFFFF. "
   seed := self peekInteger.
   ^ seed!
peek
```

```
   " This method answers the next random number that will be
     generated as a Float in the range [0..1). It answers the
     same value for all successive message sends. "
   ^ self peekInteger * PMmu1!
peekInteger
    " This method generates random instances of Integer
      in the interval 0 to 16r7FFFFFFF.
      This method does NOT update the seed; repeated sends answer
      the same value. The algorithm is described in detail in
      'Random Number Generators: Good Ones Are Hard to Find'
      by Stephen K. Park and Keith W. Miller,
      (Comm. Asso. Comp. Mach., 31(10):1192--1201, 1988). "
   | lo hi aLoRHi answer |
   hi     := seed quo: PMq.
   lo     := seed rem: PMq.
   aLoRHi := (PMa * lo) - (PMr * hi).
   answer := (aLoRHi > 0)
      ifTrue:  [ ^ aLoRHi ].
   ^ aLoRHi + PMm!
seed: anInteger
   seed := anInteger! !
```

Example 121, Filein

Testing the Generator

Using the default seed, the first 10 values are:

```
0.1492432697 0.3316330217 0.7561964480 0.3937015400 0.9417831814
0.5499291939 0.6599625962 0.9913545591 0.6960744326 0.9229878997
```

If you don't get these values, do not use the generator! Something is badly wrong!

Part VIII: Files and Streams

File Access

Frequently Asked Question

8.1 • Is there a platform-dependent way to access files?

Streams

Frequently Asked Question

8.2 • What is a stream?

(streams are bookkeeping aids -- compare a stream with equivalent done by hand.)◄

Frequently Asked Question

8.3 • What kinds of streams are there?

Frequently Asked Question

8.4 • What kinds of collections can be streamed over?

Frequently Asked Question

8.5 • When should a stream be used?

8.6 • Are streams better than concatenation?

Part IX: Processes and Exceptions

asdf asdf asfd asfd.

Processes

Frequently Asked Question

9.1 • What is a process?

Frequently Asked Question

9.2 • When do I need to use processes?

Frequently Asked Question

9.3 • How do I start a process?

Frequently Asked Question

9.4 • Are Smalltalk processes operating system threads?

Semaphores

Exception Handling

Part X: Tips and Tricks

Being a bag of tricks, algorithms, tips, & even some code to torment both the beginner and the experienced.

Environmental

Frequently Asked Question

10.1 • How can I know what platform I'm running on?

IBM Smalltalk

The expression in Example 122 answers ’`PM`’, ’`WIN32s`’, and other values depending on the platform.

```
    System subsystemType: ’CW’
```

Example 122, Evaluate

VisualWorks

You can use:

```
    Screen default platformName
```

to return the platform you are currently running on, or:

```
    OSHandle currentOS
```

to return the name of the operating system. Then:

```
    OSHandle currentPlatform
```

answers detailed information about the platform VW is running on.

—Thanks to: Vadim Sokolovsky, GreenPoint, Inc

10.2 • How can I get the contents of environment variables

The answer depends on the implementation.

IBM Smalltalk

Try:

```
   'path' abtScanEnv
```

or more generally, the environment name as a string is sent the message `abtScanEnv`.

This is in Smalltalk Version 3. It is not documented. I have no idea whether this is at all portable, works under Windows or on AIX, or will melt your machine.

I found this finally by executing the following code to get a list of all method names which have '`env`' in them somewhere.

```
   System allMethodsSatisfying: [ :compiledMethod |
      ('*env*' match: compiledMethod selector asString)
         ifTrue: [
            Transcript cr; show: compiledMethod printString ]
      ]
```

Example 123, Evaluate

ParcPlace VisualWorks

Digitalk VisualSmalltalk

Miscellaneous

10.3 • How can I defer calculations until they are needed?

There are a number of ways to defer a long calculation until it is needed, if ever.

Implementation One

The simplest is to create an instance variable, say `deferLongCalc`. Create a method, longCalc, whick looks like this:

```
longCalc
   deferLongCalc isNil
      ifTrue: [
         deferLongCalc := self performLongCalc ].
   ^ deferLongCalc
```

Example 124

Implementation Two

However, it would be nice to put the logic in one place, and write something like:

```
initialize
   deferLongCalc := Lazily do: [ self performLongCalc ]
longCalc
   deferLongCalc lazyValue
```

Example 125

Then the logic that checks and calculates is in some instance of `Lazily`.

There are several ways to write class `Lazily`. Possibly the simplest is shown in Example 126.

```
Object
   subclass: #Lazily
   instanceVariableNames: 'val block'
   classVariableNames: ''
   poolDictionaries: '' !

!Lazily class methods!
do: aBlock
   ^ self new setBlock: aBlock! !

!Lazily methods!
lazyValue
   val isNil
      ifTrue: [ val := block value ].
   ^ val !
setBlock: aBlock
   val := nil.
   block := aBlock! !
```

Example 126, Filein

This is, of course, a straight forward implementation of Example 128. It doesn't handle the case where a lazy value is itself a lazy value, but then it wasn't in the spec either! :-)

Implementation Three

But there are other ways, tricker and cuter if not necessarily better. The one in Example 127 might be a hair faster if, once a value is first determined, it might be used a lot, and it does handle lazy values being themselves lazy. This implementation is based on an idea of Terry Raymond proposed on `comp.lang.smalltalk`.

```
!Object methods!
lazyValue
   ^self! !

Object
  subclass: #Lazily
  instanceVariableNames: 'val'
  classVariableNames: ''
  poolDictionaries: '' !

!Lazily class methods!
do: aBlock
    | instance |
    instance := self new.
    ^ instance setValue:
        (LazyEvaluator new block: aBlock valueHolder: instance)! !

!Lazily methods!
lazyValue
   ^ val lazyValue!
setValue: aValue
   val := aValue! !

Object
  subclass: #LazyEvaluator
  instanceVariableNames: 'lazyGuy block'
  classVariableNames: ''
  poolDictionaries: '' !

!LazyEvaluator methods!
```

```
   block: aBlock valueHolder: aLazy
      lazyGuy := aLazy.
      block := aBlock!
   lazyValue
       lazyGuy setValue: block value.
       ^ lazyGuy lazyValue! !
```
Example 127, Filein

Now, here is some code to 'test' the implementation. It shows how a deferred value can itself be a deferred value.

```
" A simple test, including lazyily being lazy, nested 5 deep "
| a b |
a := Lazily do: [ 10 factorial ].
b := Lazily do: [
      Lazily do: [
         Lazily do: [
            Lazily do: [
               Lazily do: [
                  Lazily do: [
                     5 factorial ] ] ] ] ] ].
^ Array
      with: a lazyValue
      with: a lazyValue
      with: b lazyValue
" Answers: (3628800 3628800 120) "
```
Example 128, Display

Implementation Three

See [5.21] *'Is there a simple example of a good use of become:?'* on page 103 for yet another implementation.

Other Implementations

Yet another version, proposed on `comp.lang.smalltalk`, is to use the method `changeClassToThatOf:` (VisualWorks only) to change the class of an object rather than use `become:`. An implementation is left up to the reader.

VisualWorks also implements `Blockvalue>>with:` which does all this and more, including dependent values. It performance may not be as high, but one rarely cares anyhow, but it's there, it works, and it has more features.

Part XI: Writing Smalltalk Code

Writing Classes

Frequently Asked Question

11.1 • What does subclassResponsibility mean, and what is it used for?

The message `subclassResponsibility` causes an error message to be issued that says something like:

```
'My subclass should have implemented this message.'
```

It is used when a superclass needs to define some protocol but cannot provide an implementation. It documents an interface indicating that the method exists and it serves as documentation to implementers of subclasses that the method must be implemented. It is never intended that it be executed.

An example is < (less-than) in `Magnitude`. This message must be provided by all subclasses; a meaningful less-than comparison is what makes a magnitude be a magnitude. However, the actual implementation is very dependent on the data formats of the subclasses; thus `Magnitude` cannot do anything better than issue an error message.

Singletons

Accessor Methods

Frequently Asked Question

11.2 • What are accessor methods?

Frequently Asked Question

11.3 • When should accessor methods be used?

Frequently Asked Question

11.4 • Don't accessor methods expose what encapsulation wants to hide?

Frequently Asked Question

11.5 • Should accessor methods be used for every access?

(rel war)

Frequently Asked Question

11.6 • Should accessor methods always correspond to just one variable?

Frequently Asked Question

11.7 • Why is everyone screaming? What did I say??

Questions about Polymorphism

Frequently Asked Question

11.8 • What is polymorphism?

Frequently Asked Question

11.9 • Does polymorphism interact with inheritence?

Frequently Asked Question

11.10 • How can I take advantage of polymorphism

Frequently Asked Question

11.11 • What are static and dynamic polymorphism?

The word polymorphism is borrowed from the old Greek. It means something like 'multiple forms'. Polymorphism of an entity means that this entity can take on different forms over its lifetime.If it is a variable then the variable can stand for different objects at different times.

However, neither of the terms is usually applied to Smalltalk, but might be used to contrast one object-oriented language with another.

Static Polymorphism

Multiple meanings resolved statically (ie, without execution of the program being of any assistance).

Dynamic Polymorphism

Multiple meanings resolved dynamically (ie, with execution of the program assisting in the determination of which meaning to select.)

Smalltalk is dynamically polymorphic.

Double Dispatching

Getting Rid of ifTrue:ifFalse:

Frequently Asked Question

11.12 • Why do I need to get rid of ifTrue:ifFalse:?

Frequently Asked Question

11.13 • How do I get rid of ifTrue:ifFalse:?

Frequently Asked Question

11.14 • How does polymorphism get rid of ifTrue:ifFalse:?

Frequently Asked Question

11.15 • How does double dispatching get rid of ifTrue:ifFalse:?

Frequently Asked Question

11.16 • How can tables get rid of ifTrue:ifFalse:?

(Use index to table rather than testing index)

Coding Conventions

Frequently Asked Question

11.17 • When and where should comments be used?

Frequently Asked Question

11.18 • When should extra parentheses be used?

Frequently Asked Question

11.19 • How big should methods be?

Frequently Asked Question

11.20 • How big is too big?

Frequently Asked Question

11.21 • In general, how are methods indented?

Frequently Asked Question

11.22 • How are long messages indented?

Frequently Asked Question

11.23 • (ifTrue:ifFalse:, whenTrue:, ...)

Frequently Asked Question

11.24 • How are cascaded messages indented?

Frequently Asked Question

11.25 • How many statements should go on a line?

Frequently Asked Question

11.26 • When should blank lines be used?

Frequently Asked Question

11.27 • When should class names be hardcoded?

Frequently Asked Question

11.28 • How deep should control structures be nested?

Frequently Asked Question

11.29 • Where can I find out more about coding conventions?

Modifying the System

Frequently Asked Question

11.30 • Should my application include modifications to the system?

(no; keep new stuff in app's own classes;

don't modify String to add upcaseFirstLetterOfWord,

write it as an app method & pass string)

Frequently Asked Question

11.31 • Why not?

(Defensive programming)

Frequently Asked Question

11.32 • If I really need to, how do I best modify a system-provided class?

(new, private methods; new subclasses)

Part XII: Implementation

The Virtual Machine

Frequently Asked Question

12.1 • What is the virtual machine?

Frequently Asked Question

12.2 • Where I can find a description of the Smalltalk virtual machine ?

Most vendors keep their implementations a secret. The best info available is in:

- *Smalltalk-80: The Langauge and Its Implementation*, Goldberg & Robson, Addison-Wesley.
- *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, Editor, Addison-Wesley.
- GNU Smalltalk, an implementation based on Goldberg & Robson, is free, and has source. See:

  URL: `ftp://prep.ai.mit.edu/pub/gnu/`
- Little Smalltalk, a simpler implementation, is also available in source:

  URL: `ftp://ftp.cs.orst.edu/users/b/budd/little/index.html`
- *The Design and Evaluation of a High Performance Smalltalk System*; Ungar, David Michael; MIT Press, 1987. ISBN 0-262-21010-X. 288 pages. This book has various details, including how a generation scavanging garbage collector works.
- With regard to specific vendors, look in the manuals where it talks about how objects look when passed to C programs. There is typically not much detail, but one can get some hints. In VisualWorks, the source for the compiler is provided with the development environment, and will be full of hints about how things

work.

- The proceedings of OOPSLA has relevant papers.

Optimizing

Frequently Asked Question

12.3 • How can Smalltalk be optimized?

Note that Smalltalk compilers are free to cheat like mad. It all is in line with Peter Deutsch's observation that implementing a language like Smalltalk efficiently requires the implementor to cheat, but that's okay as long as you don't get caught. If the compiler wants to play games with variables that might hold integers, it can -- so long as it doesn't break the language semantics. And all Smalltalk systems DO play games with integers.

more more more more◄

Questions about Garbage Collection

Frequently Asked Question

12.4 • What is garbage collection?

Frequently Asked Question

12.5 • Isn’t garbage collection slow?

Frequently Asked Question

12.6 • Can garbage collection be turned on or off?

Garbage collection is an integral part of Smalltalk. It is always on and can not be turned off or bypassed.

Frequently Asked Question

12.7 • Won’t garbage collection cause long pauses?

Frequently Asked Question

12.8 • Does garbage collection really make coding simpler?

Frequently Asked Question

12.9 • How does garbage collection work?

Frequently Asked Question

12.10 • What is Generation Scavenging?

Frequently Asked Question

12.11 • How do I force garbage collection to happen?

The technique varies by implementation.

VisualWorks

IBM Smalltalk

Send either of these two messages:

```
" IBM Smalltalk: Force garbage collection "
System globalGarbageCollect
```

Example 129, Evaluate

```
" IBM Smalltalk: Force garbage collection "
System scavenge
```

Example 130, Evaluate

Digitalk

VMARK Enfin Smalltalk

Frequently Asked Question

12.12 • How can I assure that critical bits of code won’t cause garbage collection?

Metaclasses

Frequently Asked Question

12.13 • Where is the 'new' method (or: Why is 'new' THERE!)?

Simply put, classes inherit from instance methods of `Class`, which inherits from `Behavior`, which inherits from `Object`.

First off, browsers lie: classes do *not* have 'instance methods' and 'class methods'. There is only one kind of method, not two.

There are *two* kinds of instances, however: ones than can themselves have instances, and ones that cannot. Those that can have instances are called classes (or metaclasses, sometimes), and those that cannot are just called instances (to add to the confusion!)

Inheritence for instances that are classes is different than for instances that are not classes. Non-class instances have the inheritence hierarchy you learned at your mother's knee:

```
   Object
      Magnitude
         Number
            Float
            Integer
         Date
      ...
```

Instances that are classes have an extended hierarchy, as shown below. (Some implementations have another class between `Behavior` and `Class`.)

```
      Object
         Behavior
      _____ Class _________
               Object class
                  Magnitude class
                     Number class
                        Float class
                        Integer class
                     Date class
```

It is the classes above the line that make classes act like classes.

If classes are instances, what are they instances of? Well, they are instances of some class, as are all objects. But then what are *those* classes instances of?? Well, they are instances of a class too! This could go on forever (as do the meta-genies in *Godel-Escher-Bach*) but they don't. Here's what happens.

Using <= to mean *instance-of*, the relationship between class `String` and one of its instances can be written like this:

```
      aString <= String
```

This is read: '`aString` is an instance of class `String`'. Taking this one step more:

```
      aString <= String <= String class
```

This means that class `String` is an instance of a class named '`String class`'. You've probably noted this already if you ever evaluated one of the expressions:

```
      'asdf' class class
```

or:

```
      String class
```

and got the answer:

```
String class
```

If you carry this one step further and ask:

```
String class class
```

you will get:

```
Metaclass
```

Thus:

```
aString <= String <= String class <= Metaclass
```

So what is `Metaclass` (spelled `MetaClass` on some systems)? It is another subclass of `Behavior`. (Again, sometimes there are additional classes mixed in here or there under `Behavior`.)

```
Object
   Behavior
      Class
      Metaclass
```

But `Metaclass` is a class, so if we keep asking:

Expression	Result
`String class class class`	`Metaclass class`
`String class class class class`	`Metaclass`
`String class class class class class`	`Metaclass class`

Which means:

```
aString <= String <= String class <= Metaclass
   <= Metaclass class <= Metaclass <= Metaclass class ...
```

Lets look at a bigger picture. In the table below, the bottom line is an instance-of expression similar to the one just above. Stacked above each element in the diagram are the superclasses; read this upwards. Thus, `String` inherits from `Collection` (at least in our simplified view), which inherits from `Object`, which 'inherits from `nil`', or doesn't inherit from anything.

```
                 nil                               nil
                 Object                            Object
      nil        Class             nil             Class
      Object     Object class      Object          Object class
      Collection Collection class  Class           Class class
 'z' <= String    <= String class    <= Metaclass <= Metaclass class
```

There are two anomolies here:

(1) That class `Object class` inherits from class `Class`. (Try reading that aloud!)

(2) That class `Metaclass` is an instance of an instance of itself!

These two anomolies allow Smalltalk to have a clean, elegant, and (after some contemplation) understandable self-definitional facility in which everything is an object and all objects are instances of classes (including classes themselves).

So, now the *Big Lie* (George Bosworth calls it an *Artful Deception*) should be clearer. Classes do not have two kinds of methods. They have just one kind of method, and the methods are for the *instances* of that class:

- Methods that belong to `String` are for instances of `String`.
- Methods that belong to `String class` are for (the sole) instance of class `String class`.

Metaclasses (instances of class `Metaclass`) exist so that (among other reasons) there is somewhere to put the methods of its instance. There is just one such instance per class, since each class has the need to hold just one set of methods.

There is thus a direct correspondence between a class and its metaclass. The table below, where classes are indented to show their hierarchical relationship, illustrates this correspondence:

```
Object          <= Object class         <= Metaclass
 Behavior       <=  Behavior class      <= Metaclass
  Metaclass     <=   Metaclass class    <= Metaclass
  Class         <=   Class class        <= Metaclass
 Magnitude      <=  Magnitude class     <= Metaclass
  Number        <=   Number class       <= Metaclass
   Integer      <=    Integer class     <= Metaclass
  Date          <=   Date class         <= Metaclass
```

and so on.

It *should* always be `true` that:

```
      Metaclass allInstances size = (Object class allSubclasses + 1)
```

(In IBM Smalltalk there can be anomolies if a class has been deleted but a browser is still open that references it in its hierarchy list; updating the list will correct this. Other systems may have analagous anomolies. Your mileage may vary.)

So why is `new` in `Behavior`? Because `Behavior` describes the mechanics of *being* a class (and being a metaclass) and of being instances, and thus that is where it belongs!

Questions on Object Pointers

Frequently Asked Question

12.14 • What is an object pointer?

Frequently Asked Question

12.15 • What things are encoded in an object pointer?

Part XIII: Development Tools

Using Workspaces

Frequently Asked Question

13.1 • How do I evaluate code in a workspace?

Frequently Asked Question

13.2 • Does code evaluated in a workspace act exactly like code in a method?

Frequently Asked Question

13.3 • Should I try things first in a workspace?

Frequently Asked Question

13.4 • How can I time code in a workspace?

Frequently Asked Question

13.5 • Are there any tricks for effective code timing?

Frequently Asked Question

13.6 • How does code in a workspace actually get executed?

Frequently Asked Question

13.7 • What context does evaluated code run in?

Browsing

Frequently Asked Question

13.8 • How should categories be used?

Inspecting

Frequently Asked Question

13.9 • What is the fastest way to run some code?

(evaluate in workspace...)

Frequently Asked Question

13.10 • What is an inspector?

Frequently Asked Question

13.11 • How can an inspector be opened for an object?

Debugging

Frequently Asked Question

13.12 • What is the debugger?

Frequently Asked Question

13.13 • How can the debugger be opened?

Frequently Asked Question

13.14 • Why don't I see every statement when stepping?

Profiling

Frequently Asked Question

13.15 • Does Smalltalk have a code profiler?

System Interfaces

Frequently Asked Question

13.16 • Can garbage collection be forced to take place?

[12.11] *'How do I force garbage collection to happen?'* on page 169.

Frequently Asked Question

13.17 • How does a program tell how much memory is available?

Frequently Asked Question

13.18 • Can command line parameters be passed to Smalltalk?

Frequently Asked Question

13.19 • Can the image be saved from a program?

Other Tools

Frequently Asked Question

13.20 • Does Smalltalk have an equivalent to LEX and YACC.

There are several answers:

1. The T-Gen package provides similar function. The original version of T-gen was written for ObjectWorks Smalltalk. It has been ported to Digitalk Smalltalk and IBM Smalltalk. It is available from the University of Illinois Smalltalk Archive:

   ```
   ftp://st.cs.uiuc/pub/st80_r41/T-gen2.1/
   ```

2. In VisualWorks, look at the `Scanner` and `Parser` classes. Both `Scanner` and `Parser` are intended for use handling code (for languages like Smalltalk). They may not be exactly what you want for languages that differ or that generate data.

Frequently Asked Question

13.21 • What is HotDraw?